

# FEW-SHOT IN-CONTEXT PREFERENCE LEARNING USING LARGE LANGUAGE MODELS

Anonymous authors

Paper under double-blind review

## ABSTRACT

Designing reward functions is a core component of reinforcement learning but can be challenging for truly complex behavior. Reinforcement Learning from Human Feedback (RLHF) has been used to alleviate this challenge by replacing a hand-coded reward function with a reward function learned from preferences. However, it can be exceedingly inefficient to learn these rewards as they are often learned tabula rasa. We investigate whether Large Language Models (LLMs) can reduce this query inefficiency by converting an iterative series of human preferences into code representing the rewards. We propose In-Context Preference Learning (ICPL), a method that uses the grounding of an LLM to accelerate learning reward functions from preferences. ICPL takes the environment context and task description, synthesizes a set of reward functions, and then repeatedly updates the reward functions using human feedback over videos of the resultant policies over a small number of trials. Using synthetic preferences, we demonstrate that ICPL is orders of magnitude more efficient than RLHF and is even competitive with methods that use ground-truth reward functions instead of preferences. Finally, we perform a series of human preference-learning trials and observe that ICPL extends beyond synthetic settings and can work effectively with humans-in-the-loop.

## 1 INTRODUCTION

Reward functions are a critical component of reinforcement learning (RL). However, specifying these functions becomes increasingly challenging as the complexity of the desired tasks grows. Recent advancements in pretrained foundation models have inspired approaches that leverage large language models to synthesize reward functions from task descriptions (Yu et al., 2023a; Ma et al., 2024; Yu et al., 2023b). Despite these innovations, existing methods still depend on human-designed sparse rewards or task-specific metrics to construct the reward functions. This is challenging for tasks where we cannot define any clear reward signals as the task is primarily semantically defined. For example, it is tricky to write down a reward function for a humanoid robot that corresponds to "moving like a human".

Preference-based RL offers a potential solution to this problem. Instead of relying on a human to write the reward function, we learn a reward model based on human preferences across different trajectories. This interactive approach has shown success in various RL tasks, including standard benchmarks (Christiano et al., 2017; Ibarz et al., 2018), encouraging novel behaviors (Liu et al., 2020; Wu et al., 2021), and overcoming reward exploitation (Lee et al., 2021a). However, in more complex tasks requiring extensive agent-environment interactions, preference-based RL often necessitates hundreds or even thousands of human queries to provide effective feedback. This is likely because the reward models are typically learned tabula rasa. For instance, a robotic arm button-pushing task requires over 10k queries to learn reasonable behavior (Lee et al.), which can be a major bottleneck.

In this work, we introduce a novel method, In-Context Preference Learning (ICPL), which significantly enhances the sample efficiency of preference-based RL through LLM guidance. Our primary insight is to harness the coding capabilities of LLMs to autonomously generate reward functions, then utilize human preferences through in-context learning to refine these functions. Specifically, ICPL leverages an LLM, such as GPT-4, to generate executable, diverse reward functions based on

the task description and environment source code. We acquire preferences by evaluating the agent behaviors resulting from these reward functions, selecting the most and least preferred behaviors. The selected functions, along with historical data such as reward traces of the generated reward functions from RL training, are then fed back into the LLM to guide subsequent iterations of reward function generation. We hypothesize that as a result of its grounding in text data, ICPL will be able to improve the quality of the reward function through incorporating the preferences and the history of the generated reward functions, ensuring they align more and more closely with human preferences. Unlike evolutionary search methods like EUREKA Ma et al. (2023), there is no ground-truth reward function that the LLM can use to evaluate agent performance, and thus, success here would demonstrate that LLMs have some native preference-learning capabilities.

To study the effectiveness of ICPL, we perform experiments on a diverse set of RL tasks. For scalability, we first study tasks with synthetic preferences where a ground-truth reward function is used to assign preference labels. We observe that compared to traditional preference-based RL algorithms, ICPL achieves over a 30 times reduction in the required number of preference queries to achieve equivalent or superior performance. **Moreover, ICPL achieves performance comparable to reward-generation methods that utilize a ground truth sparse reward as feedback (Ma et al., 2023).** Finally, we test ICPL on a particularly challenging task, “making a humanoid jump like a real human,” where designing a reward is difficult. By using real human feedback, our method successfully trained an agent capable of bending both legs and performing stable, human-like jumps, showcasing the potential of ICPL in tasks where human intuition plays a critical role.

In summary, the contributions of the paper are the following:

- We propose ICPL, an LLM-based preference learning algorithm. Over a synthetic set of preferences, we demonstrate that ICPL can iteratively output rewards that increasingly reflect preferences. Via a set of ablations, we demonstrate that this improvement is relatively monotonic, suggesting that preference learning is occurring as opposed to a random search.
- We demonstrate, via human-in-the-loop trials, that ICPL is able to work effectively with humans-in-the-loop despite significantly noisier preference labels.
- We demonstrate that ICPL sharply outperforms tabula-rasa RLHF methods and is also competitive with methods that rely on access to a ground-truth reward.

## 2 RELATED WORK

**Reward Design.** In reinforcement learning, reward design is a core challenge, as most rewards both represent a desired set of behaviors and provide enough signal for learning. The most common approach to reward design is handcrafting, which requires a large number of trials by experts (Sutton, 2018; Singh et al., 2009). Since hand-coded reward design requires extensive engineering effort, several prior works have studied modeling the reward function with precollected data. For example, Inverse Reinforcement Learning (IRL) aims to recover a reward function from expert demonstration data (Arora & Doshi, 2021; Ng et al., 2000). With advances in pretrained foundation models, some recent works have also studied using large language models or vision-language models to provide reward signals (Ma et al., 2022; Fan et al., 2022; Du et al., 2023; Karamchetti et al., 2023; Kwon et al., 2023; Wang et al., 2024; Ma et al., 2024; Holk et al., 2024). Among these approaches, EUREKA (Ma et al., 2023) is the closest to our work, instructing the LLM to generate and select novel reward functions based on environment feedback with an evolutionary framework. However, **EUREKA’s primary goal is to test whether LLMs can produce better reward functions than humans by leveraging human-designed sparse rewards as fitness scores to evolve reward functions.** In contrast, **ICPL is designed for tasks even without available sparse rewards and leverages LLM grounding to accelerate learning reward functions directly from human preferences.** We note that EUREKA also has a small, preliminary investigation combining human preferences with an LLM to generate human-preferred behaviors in a single scenario. **Our approach relies solely on preferences, yielding higher human-involvement efficiency.** This paper is a significantly scaled-up version of that investigation as well as a methodological study of how best to incorporate prior rounds of feedback.

**Human-in-the-loop Reinforcement Learning.** Feedback from humans has been proven to be effective in training reinforcement learning agents that better match human preferences (Retzlaff et al., 2024; Mosqueira-Rey et al., 2023; Kwon et al., 2023). Previous works have investigated human

108 feedback in various forms, such as trajectory comparisons, preferences, demonstrations, and corrections  
 109 (Wirth et al., 2017; Ng et al., 2000; Jeon et al., 2020; Peng et al., 2024). Among these various  
 110 methods, preference-based RL has been successfully scaled to train large foundation models for hard  
 111 tasks like dialogue, e.g. ChatGPT (Ouyang et al., 2022). In LLM-based applications, prompting is  
 112 a simple way to provide human feedback in order to align LLMs with human preferences (Giray,  
 113 2023; White et al., 2023; Chen et al., 2023). Iteratively refining the prompts with feedback from  
 114 the environment or human users has shown promise in improving the output of the LLM (Wu et al.,  
 115 2021; Nasiriany et al., 2024). This work extends the usage of the ability to control LLM behavior  
 116 via in-context prompts. We aim to utilize interactive rounds of preference feedback between the  
 117 LLM and humans to guide the LLM to generate reward functions that can elicit behaviors that align  
 118 with human preferences.

### 119 3 PROBLEM DEFINITION

120 Our goal is to design a reward function that can be used to train reinforcement learning agents that  
 121 demonstrate human-preferred behaviors. It is usually hard to design proper reward functions in  
 122 reinforcement learning that induce policies that align well with human preferences.  
 123

124 **Markov Decision Process with Preferences** (Wirth et al. (2017)) A *Markov Decision Process with Preferences* (MDPP) is defined as a tuple  $M = \langle \mathcal{S}, \mathcal{A}, \mu, \sigma, \gamma, \rho \rangle$  where  $\mathcal{S}$  denotes the state space,  
 125  $\mathcal{A}$  denotes the action space,  $\mu$  is the distribution of initial states,  $\sigma$  is the state transition model,  
 126  $\gamma \in [0, 1]$  is the discount factor.  $\rho$  is the preference relation over trajectories, i.e.  $\rho(\tau_i \succ \tau_j)$   
 127 denotes the probability with which trajectory  $\tau_i$  is preferred over  $\tau_j$ . Given a set of preferences  $\zeta$ ,  
 128 the goal in an MDPP is to find a policy  $\pi^*$  that maximally complies with  $\zeta$ . A preference  $\tau_1 \succ \tau_2$  is  
 129 satisfied by  $\pi$  if and only if  $\Pr_{\pi}(\tau_1) > \Pr_{\pi}(\tau_2)$  where  $\Pr_{\pi}(\tau) = \mu(s_0) \prod_{t=0}^{|\tau|} \pi(a_t | s_t) \sigma(s_{t+1} | s_t, a_t)$ .  
 130 This can be viewed as finding a  $\pi^*$  that minimizes a preference loss  $L(\pi_{\zeta}) = \sum_i L(\pi, \zeta_i)$ , where  
 131  $L(\pi, \tau_1 \succ \tau_2) = -(\Pr_{\pi}(\tau_1) - \Pr_{\pi}(\tau_2))$ .  
 132

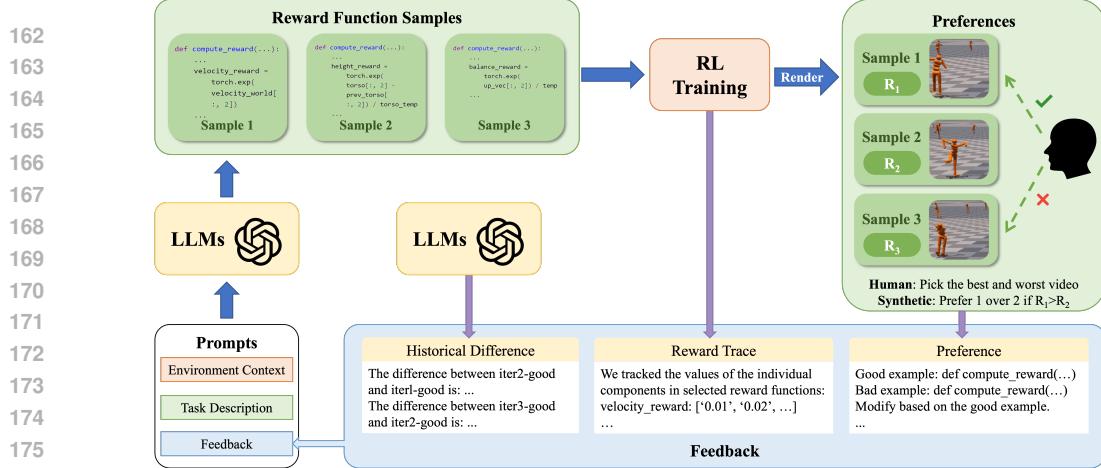
133 **Reward Design Problem with Preferences.** A *reward design problem with preferences* (RDPP)  
 134 is a tuple  $P = \langle M, \mathcal{R}, A_M, \zeta \rangle$ , where  $M$  is a Markov Decision Process with Preferences,  $\mathcal{R}$  is the  
 135 space of reward functions,  $A_M(\cdot) : \mathcal{R} \rightarrow \Pi$  is a learning algorithm that outputs a policy  $\pi$  that  
 136 optimizes a reward  $R \in \mathcal{R}$  in the MDPP.  $\zeta = \{(\tau_1, \tau_2)\}$  is the set of preferences. In an RDPP,  
 137 the goal is to find a reward function  $R \in \mathcal{R}$  such that the policy  $\pi = A_M(R)$  that optimizes  $R$   
 138 maximally complies with the preference set  $\zeta$ . In Preference-based Reinforcement Learning, the  
 139 learning algorithms usually involve multiple iterations, and the preference set  $\zeta$  is constructed in  
 140 every iteration by sampling trajectories from the policy or policy population.  
 141

### 142 4 METHOD

143 Our proposed method, In-Context Preference Learning (ICPL), integrates LLMs with human preferences to synthesize reward functions. The LLM receives environmental context and a task description to generate an initial set of  $K$  executable reward functions. ICPL then iteratively refines these functions. In each iteration, the LLM-generated reward functions are trained within the environment, producing a set of agents; we use these agents to generate videos of their behavior. A ranking is formed over the videos, from which we retrieve the best and worst reward functions corresponding to the top and bottom videos in the ranking. These selections serve as examples of positive and negative preferences. The preferences, along with additional contextual information, such as reward traces and differences from previous good reward functions, are provided as feedback prompts to the LLM. The LLM takes in this context and is asked to generate a new set of rewards. Algo. 1 presents the pseudocode, and Fig. 1 illustrates the overall process of ICPL.  
 144

#### 145 4.1 REWARD FUNCTION INITIALIZATION

146 To enable the LLM to synthesize effective reward functions, it is essential to provide task-specific  
 147 information, which consists of two key components: a description of the environment, including  
 148 the observation and action space, and a description of the task objectives. At each iteration, ICPL  
 149 ensures that  $K$  executable reward functions are generated by resampling until there are  $K$  executable  
 150 reward functions.  
 151



177 Figure 1: ICPL employs the LLM to generate initial  $K$  executable reward functions based on the  
178 task description and environment context. Using RL, agents are trained with these reward functions.  
179 Videos are generated of the resultant agent behavior from which human evaluators select their most  
180 and least preferred. These selections serve as examples of positive and negative preferences. The  
181 preferences, along with additional contextual information, are provided as feedback prompts to the  
182 LLM, which is then requested to synthesize a new set of reward functions. For experiments simulating  
183 human evaluators, task scores are used to determine the best and worst reward functions.

#### Algorithm 1: In-Context Preference Learning (ICPL)

---

**Input:** Number of iterations  $N$ , Number of samples  $K$ , Environment Env, Coding LLM  $LLM_{RF}$   
 // Initialize the prompt with environment context and task description

- 1 Prompt  $\leftarrow$  InitializePrompt(Env)
- 2 **for**  $i \leftarrow 1$  **to**  $N$  **do**
- 3      $RF_1, \dots, RF_K \leftarrow LLM_{RF}(\text{Prompt}, K)$   
      // Render videos for each reward function
- 4      $Video_1, \dots, Video_K \leftarrow \text{Render}(\text{Env}, RF_1), \dots, \text{Render}(\text{Env}, RF_K)$   
      // Human selects the most preferred (G) and least preferred (B) videos
- 5      $G, B \leftarrow \text{Human}(Video_1, \dots, Video_K)$   
      // Retrieve the best and worst reward functions
- 6      $GoodRF, BadRF \leftarrow RF_G, RF_B$   
      // Update the prompt with feedback
- 7      $\text{Prompt} \leftarrow GoodRF + BadRF + \text{HistoricalDifference} + \text{RewardTrace}$
- 8 **end**

---

## 4.2 SEARCH REWARD FUNCTIONS BY HUMAN PREFERENCES

For tasks without reward functions, the traditional preference-based approach typically involves constructing a reward model, which often demands substantial human feedback. Our approach, ICPL, aims to enhance efficiency by leveraging LLMs to directly search for optimal reward functions without the need to learn a reward model. To expedite this search process, we use an LLM-guided search to find well-performing reward functions. Specifically, we generate  $K = 6$  executable reward functions per iteration across  $N = 5$  iterations. In each iteration, humans select the most preferred and least preferred videos, resulting in a good reward function and a bad one. These are used as a context for the LLM to use to synthesize a new set of  $K$  reward functions. These reward functions are then used in a PPO (Schulman et al., 2017) training loop, and videos are rendered of the final trained agents.

## 4.3 AUTOMATIC FEEDBACK

In each iteration, the LLM not only incorporates human preferences but also receives automatically synthesized feedback. This feedback is composed of three elements: the evaluation of selected reward functions, the differences between historical good reward functions, and the reward trace of these historical reward functions.

**Evaluation of reward functions:** The component values that make up the good and bad reward functions are obtained from the environment during training and provided to the LLM. This helps the LLM assess the usefulness of different parts of the reward function by comparing the two.

**Differences between historical reward functions:** The best reward functions selected by humans from each iteration are taken out, and for any two consecutive good reward functions, their differences are analyzed by another LLM. These differences are supplied to the primary LLM to assist in adjusting the reward function.

**Reward trace of historical reward functions:** The reward trace, consisting of the values of the good reward functions during training from all prior iterations, is provided to the LLM. This reward trace enables the LLM to evaluate how well the agent is actually able to optimize those reward components.

## 5 EXPERIMENTS

In this section, we conducted two sets of experiments to evaluate the effectiveness of our method: one using proxy human preferences and the other using real human preferences.

1) Proxy Human Preference: In this experiment, human-designed rewards, taken from EU-REKA (Ma et al., 2023), were used as proxies of human preferences. Specifically, if ground truth reward  $R_1 > R_2$ , sample 1 is preferred over sample 2. This method enables rapid and quantitative evaluation of our approach. It corresponds to a noise-free case that is likely easier than human trials; if ICPL performed poorly here it would be unlikely to work in human trials. Importantly, human-designed rewards were only used to automate the selection of samples and were not included in the prompts sent to the LLM; the LLM **never observes the functional form of the ground truth rewards nor does it ever receive any values from them**. Since proxy human preferences are free from noise, they offer a reliable comparison to evaluate our approach efficiently. However, as discussed later in the limitations section, these proxies may not correctly measure challenges in human feedback such as inability to rank samples, intransitive preferences, or other biases.

2) Human-in-the-loop Preference: To further validate our method, we conducted a second set of experiments with human participants. These participants repeated the tasks from the Proxy Human Preferences and engaged in an additional task that lacked a clear reward function: “Making a humanoid jump like a real human.”

### 5.1 TESTBED

All experiments were conducted on tasks from the Eureka benchmark (Ma et al., 2023) based on IsaacGym, covering a diverse range of environments: *Cartpole*, *BallBalance*, *Quadcopter*, *Anymal*, *Humanoid*, *Ant*, *FrankaCabinet*, *ShadowHand*, and *AllegroHand*. We adhered strictly to the original task configurations, including observation space, action space, and reward computation. This ensures that our method’s performance was evaluated under consistent and well-established conditions across a variety of domains.

### 5.2 BASELINES

We consider three preference-based RL methods as baselines, which update reward models during training. B-Pref (Lee et al.), a benchmark specifically designed for preference-based reinforcement learning, provides two of our baseline algorithms: **PrefPPO** and **PEBBLE**. PrefPPO is based on the on-policy RL algorithm PPO, while PEBBLE builds upon the off-policy RL algorithm SAC. Additionally, we include **SURF** (Park et al., 2022), which enhances PEBBLE by utilizing unlabeled samples with data augmentation to improve feedback efficiency. For each task, we use the default hyperparameters of PPO and SAC provided by IsaacGym, which were fine-tuned for high performance. This ensures a fair comparison across methods. Further details can be found in Appendix A.3.

### 5.3 EXPERIMENT SETUP

**Training Details.** We trained policies and rendered videos on a single A100 GPU machine. The total time for a full experiment was less than one day of wall clock time. We utilized GPT-4,

Table 1: The final task score of all methods across different tasks in IssacGym. The top result and those within one standard deviation are highlighted in bold. Standard deviations are provided in Table 6 of Appendix A.5.1 due to space limitations.

	Cart.	Ball.	Quad.	Anymal	Ant	Human.	Franka	Shadow	Allegro
PrefPPO-49	<b>499</b>	<b>499</b>	-1.066	-1.861	0.743	0.457	0.0044	0.0746	0.0125
<b>PEBBLE-49</b>	<b>499</b>	<b>499</b>	-1.190	-1.521	5.9891	0.903	0.0453	0.2142	0.1467
<b>SURF-49</b>	<b>499</b>	<b>499</b>	-1.208	-1.35	0.815	1.675	0.0039	0.1500	0.1116
PrefPPO-15k	<b>499</b>	<b>499</b>	-0.250	-1.357	4.626	1.317	0.0399	0.0468	0.0157
<b>PEBBLE-15k</b>	<b>499</b>	<b>499</b>	-0.231	-0.730	8.543	4.074	0.6089	0.2438	0.2401
<b>SURF-15k</b>	<b>499</b>	<b>499</b>	-0.266	-0.346	7.859	3.292	0.3434	0.2145	0.2352
ICPL(Ours)	<b>499</b>	<b>499</b>	<b>-0.0195</b>	<b>-0.007</b>	<b>12.04</b>	<b>9.227</b>	<b>0.9999</b>	<b>13.231</b>	<b>25.030</b>
Eureka	499	499	-0.023	-0.003	10.86	9.059	0.9999	11.532	25.250

specifically the GPT-4-0613, as the backbone LLM in the Proxy Human Preference experiment. For the Human-in-the-loop Preference experiment, we employ GPT-4o.

**Evaluation Metric.** Here, we provide a specific explanation of how sparse rewards (detailed in Appendix A.4) are used as task metrics in the adopted IsaacGym tasks. The task metric is the average of the sparse rewards across **parallel environments**. To assess the generated reward function or the learned reward model for each RL run, we take the **maximum task metric value sampled at fixed intervals, marked as *task score of reward function/model* (RTS)**. In each iteration, ICPL generates 6 RL runs and selects the highest RTS as the result for that iteration. ICPL performs 5 iterations and then selects the highest RTS from these iterations as the **task score (TS)** for each **experiment**. Due to the inherent randomness of LLMs, we run 5 experiments for all methods, and report the highest TS as the **final task score (FTS)** for each approach. A higher FTS indicates better performance across all tasks.

## 5.4 RESULTS OF PROXY HUMAN PREFERENCE

### 5.4.1 MAIN RESULTS

In ICPL, we use human-designed sparse rewards as proxies to simulate ideal human preferences. Specifically, in each iteration, we select the reward function with the highest RTS as the good example and the reward function with the lowest RTS as the bad example for feedback. All baseline methods leverage dense rewards to simulate proxy human preference, offering a stronger and more informative signal for labeling preferences. If the cumulative dense reward of trajectory 1 is greater than that of trajectory 2, then trajectory 1 is preferred over trajectory 2. We also tried sparse rewards as proxy human preference in baseline methods and observed similar performance. Table 1 shows the final task score (FTS) for all methods across IsaacGym tasks.

For ICPL and baselines, we track the number of synthetic queries  $Q$  required as a proxy for measuring the likely real human effort involved, which is crucial for methods that rely on human-in-the-loop preference feedback. Specifically, we define a single query as a human comparing two trajectories and providing a preference. In ICPL, each iteration generates  $K$  reward function samples, resulting in  $K$  corresponding videos. The human compares these videos, first selecting the best one, then picking the worst from the remaining  $K - 1$  videos. After  $N = 5$  iterations, the best video of each iteration is compared to select the overall best. The number of human queries  $Q$  can be calculated as  $Q = (K - 1) \times 2N - 1$ . For ICPL, with  $K = 6$  and  $N = 5$ , this results in  $Q = 49$ . In baselines, the simulated human teacher compares two sampled trajectories and provides a preference label to update the reward model. We set the maximum number of queries to  $Q = 49$ , matching ICPL, and also test  $Q = 15k$ , denoted as Baseline-# $Q$  in Table 1, to compare the final task score (FTS) across different tasks. **Additional results with  $Q = 150, 1.5k$  can be found in Table 6 of Appendix A.5.1.**

As shown in Table 1, for the simpler tasks like *Cartpole* and *BallBalance*, all methods achieve equal performance. Notably, we observe that for these particularly simple tasks, ICPL can generate correct reward functions in a zero-shot manner, without requiring feedback. As a result, ICPL only requires querying the human 5 times, while baseline methods, after 5 queries, fail to train a reasonable reward model with the preference-labeled data. For relatively more challenging tasks, Baseline-49 performs significantly worse than ICPL when using the same number of human queries. In fact, Baseline-49 fails in most tasks. As the number of human queries increases, baselines' performance improves

Table 2: Ablation studies on ICPL modules. The runs have fairly high variance so we highlight the top two results in bold. The full table with std. deviations included can be found in Appendix A.5.1. We observe that ICPL with all of the components is consistently the best performing, suggesting that most of the components are useful.

	Cart.	Ball.	Quad.	Anymal	Ant	Human.	Franka	Shadow	Allegro
ICPL w/o RT	<b>499</b>	<b>499</b>	-0.0340	-0.387	10.50	8.337	<b>0.9999</b>	10.769	<b>25.641</b>
ICPL w/o RTD	<b>499</b>	<b>499</b>	-0.0216	<b>-0.009</b>	10.53	<b>9.419</b>	<b>1.0000</b>	11.633	23.744
ICPL w/o RTDB	<b>499</b>	<b>499</b>	<b>-0.0136</b>	-0.014	<b>11.97</b>	8.214	0.5129	<b>13.663</b>	<b>25.386</b>
OpenLoop	<b>499</b>	<b>499</b>	-0.0410	-0.016	9.350	8.306	<b>0.9999</b>	9.476	23.876
ICPL(Ours)	<b>499</b>	<b>499</b>	<b>-0.0195</b>	<b>-0.007</b>	<b>12.04</b>	<b>9.227</b>	<b>0.9999</b>	<b>13.231</b>	25.030

across most tasks, but it still falls noticeably short compared to ICPL. This demonstrates that ICPL, with the integration of LLMs, can reduce human effort in preference-based learning by at least 30 times.

**Performance Analysis with Eureka** We further report Eureka’s performance (Ma et al., 2023) as an approximate upper bound on the expected performance ICPL could achieve. Eureka is an LLM-powered reward design method that uses sparse rewards as fitness scores. Specifically, the reward function with the highest RTS is selected as the candidate reward function for feedback in each iteration and RTS is incorporated as the “task score” in the reward reflection. Original Eureka generates 16 reward functions in each iteration without checking their executability, assuming at least one will typically work across all considered environments in the first iteration. To ensure a fair comparison, we modified Eureka to generate a fixed number of executable reward functions, specifically  $K = 6$  per iteration, the same as ICPL. This adjustment improves Eureka’s performance in more challenging tasks, where it often generates fewer executable reward functions. As shown in Table 1, ICPL surprisingly achieves comparable performance, indicating that ICPL’s use of LLMs for preference learning is effective.

From the analysis conducted across 7 tasks where zero-shot generation of optimal reward functions was not feasible in the first iteration, we examined which iteration’s RTS was chosen as the final FTS. The distribution of RTS selections over iterations is illustrated in Fig. 2. The results indicate that FTS selections do not always come from the last iteration; some are also derived from earlier iterations. However, the majority of FTS selections originate from iterations 4 and 5, suggesting that ICPL is progressively refining and enhancing the reward functions over successive iterations as opposed to randomly generating diverse reward functions.

### 5.5 METHOD ANALYSIS

To validate the effectiveness of ICPL’s module design, we conducted ablation studies. We aim to answer several questions that could undermine the results presented here:

1. Are components such as the reward trace or the reward difference helpful?
2. Is the LLM actually performing preference learning? Or is it simply zero-shot outputting the correct reward function due to the task being in the training data?

#### 5.5.1 ABLATIONS

The results of the ablations are shown in Table 2. In these studies, “ICPL w/o RT” refers to removing the reward trace from the prompts sent to the LLMs. “ICPL w/o RTD” indicates the removal of both the reward trace and the differences between historical reward functions from the prompts. “ICPL w/o RTDB” removes the reward trace, differences between historical reward functions, and bad reward functions, leaving only the good reward functions and their evaluation in the prompts. The “OpenLoop” configuration samples  $K \times N$  reward functions without any feedback, corresponding to the ability of the LLM to zero-shot accomplish the task.

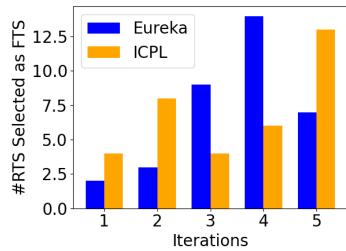


Figure 2: Distribution of which iteration is selected as the top-scoring iteration. While it is not perfectly monotonic, we observe that the final iteration is generally the best one, suggesting that the inferred reward is gradually approaching the ground-truth reward.

378 Due to the large variance of the experiments (see Appendix), we mark the top two results in bold.  
 379 As shown, ICPL achieves top 2 results in 8 out of 9 tasks and is comparable on the *Allegro* task. The  
 380 “OpenLoop” configuration performs the worst, indicating that our method does not solely rely on  
 381 GPT-4’s either having randomly produced the right reward function or having memorized the reward  
 382 function during its training. This improvement is further demonstrated in Sec. 5.5.2, where we show  
 383 the step-by-step improvements of ICPL through proxy human preference feedback. Additionally,  
 384 “ICPL w/o RT” underperforms on multiple tasks, highlighting the importance of incorporating the  
 385 reward trace of historical reward functions into the prompts.

### 386 387 5.5.2 IMPROVEMENT ANALYSIS

388 Table 1 presents the performance achieved by ICPL. While it  
 389 is possible that the LLMs could generate an optimal reward  
 390 function in a zero-shot manner, the primary focus of our anal-  
 391 ysis is not solely on absolute performance values. Rather, we  
 392 emphasize whether ICPL is capable of enhancing performance  
 393 through the iterative incorporation of preferences. We calcu-  
 394 lated the average RTS improvement over iterations relative to  
 395 the first iteration for the two tasks with the largest improve-  
 396 ments compared with “OpenLoop”, *Ant* and *ShadowHand*. As  
 397 shown in Fig. 3, the RTS exhibits an upward trend, demon-  
 398 strating its effectiveness in improving reward functions over  
 399 time. We note that this trend is roughly monotonic, indicating  
 400 that on average the LLM is using the preferences to construct  
 401 reward functions that are closer to the ground-truth reward. We  
 402 further use an example in the *Humanoid* task to demonstrate  
 403 how ICPL progressively generated improved reward functions  
 404 over successive iterations in Appendix A.5.2.

## 405 406 5.6 RESULTS OF HUMAN-IN-THE-LOOP PREFERENCE

407 To address the limitations of proxy human preferences, which simulate idealized human preference  
 408 and may not fully capture the challenges humans may face in providing preferences, we conducted  
 409 experiments with real human participants. **We recruited 7 volunteers for human-in-the-loop ex-**  
 410 **periments, with 5 assigned to IsaacGym tasks and 2 to a newly designed task. Additionally, 20**  
 411 **volunteers were recruited to evaluate the performance of different methods.** None of the volunteers  
 412 had prior experience with these tasks, ensuring an unbiased evaluation based on their preferences.

### 413 414 5.6.1 HUMAN EXPERIMENT SETUP

415 Before the experiment, each volunteer was provided with a detailed explanation of the experiment’s  
 416 purpose and process. Additionally, volunteers were fully informed of their rights, and written con-  
 417 sent was obtained from each participant. The experimental procedure was approved by the de-  
 418 partment’s ethics committee to ensure compliance with institutional guidelines on human subject  
 research.

419 In **ICPL experiments**, each volunteer was assigned an account with a pre-configured environment  
 420 to ensure smooth operation. After starting the experiment, LLMs generated the first iteration of  
 421 reward functions. Once the reinforcement learning training was completed, videos corresponding to  
 422 the policies derived from each reward function were automatically rendered. Volunteers compared  
 423 the behaviors in the videos with the task descriptions and selected both the best and the worst-  
 424 performing videos. They then entered the respective identifiers of these videos into the interactive  
 425 interface and pressed “Enter” to proceed. The human preference was processed as an LLM prompt  
 426 for generating feedback, leading to the next iteration of reward function generation.

427 This training-rendering-selection process was repeated across several iterations. At the end of the  
 428 final iteration, the volunteers were asked to select the best video from those previously marked as  
 429 good, designating it as the final result of the experiment. For IsaacGym tasks, the corresponding RTS  
 430 was recorded as TS. It is important to note that, unlike proxy human preference experiments where  
 431 the TS is the maximum RTS across iterations, in the human-in-the-loop preference experiment, TS  
 refers to the highest RTS chosen by the human, as human selections are not always based on the

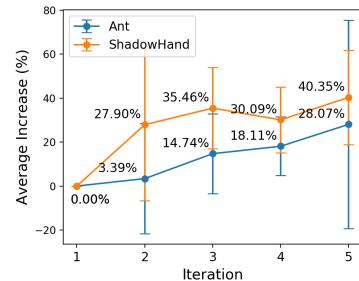


Figure 3: Average improvement of the Reward Task Score (RTS) over successive iterations relative to the first iteration in ICPL for the *Ant* and *ShadowHand* tasks, demonstrating the method’s effectiveness in refining reward functions over time.

Table 3: The final task score of human-in-the-loop preference across 5 IsaacGym tasks. The values in parentheses represent the standard deviation.

	Quadcopter	Ant	Humanoid	Shadow	Allegro
OpenLoop	-0.0410(0.32)	9.350(2.35)	8.306(1.63)	9.476(2.44)	23.876(7.91)
ICPL-proxy	-0.0195(0.09)	12.040(1.69)	9.227(0.93)	13.231(1.88)	25.030(3.72)
ICPL-real	-0.0183(0.29)	11.142(0.37)	8.392(0.53)	10.74(0.92)	24.134 (6.52)

maximum RTS at each iteration. Given that ICPL required reinforcement learning training in every iteration, each experiment lasted two to three days. Each volunteer was assigned a specific task and conducted five experiments, one for each task, with the highest TS being recorded as FTS in IsaacGym tasks.

#### 5.6.2 ISAACGYM TASKS

Due to the simplicity of the *Cartpole*, *BallBalance*, *Franka* tasks, where LLMs were able to zero-shot generate correct reward functions without any feedback, these tasks were excluded from the human trials. The *Anymal* task, which involved commanding a robotic dog to follow random commands, was also excluded as it was difficult for humans to evaluate whether the commands were followed based solely on the videos. For the 5 adopted tasks, we describe in the Appendix A.6.2 how humans infer tasks through videos and the potential reasons that may lead to preference rankings that do not accurately reflect the task.

Table 3 presents the FTS for the human-in-the-loop preference experiments conducted across 5 suitable IsaacGym tasks, labeled as “ICPL-real”. The results of the proxy human preference experiment are labeled as “ICPL-proxy”. As observed, the performance of “ICPL-real” is comparable or slightly lower than that of “ICPL-proxy” in all 5 tasks, yet it still outperforms the “OpenLoop” results in 3 out of 5 tasks. This indicates that while humans may have difficulty providing consistent preferences from videos as proxies, their feedback can still be effective in improving performance when combined with LLMs.

#### 5.6.3 HUMANOIDJUMP TASK

In our study, we introduced a new task: *HumanoidJump*, with the task description being “to make humanoid jump like a real human.” Defining a precise task metric for this objective is challenging, as the criteria for human-like jumping are not easily quantifiable. The task-specific prompts used in this experiment are detailed in the Appendix A.6.3.



Figure 4: A common behavior.

The most common behavior observed in this task, as illustrated in Fig. 4, is what we refer to as the “leg-lift jump.” This behavior involves initially lifting one leg to raise the center of mass, followed by the opposite leg pushing off the ground to achieve lift. The previously lifted leg is then lowered to extend airtime. Various adjustments of the center of mass with the lifted leg were also noted. This behavior meets the minimal metric of a jump: achieving a certain distance off the ground. If feedback were provided based solely on this minimal metric, the “leg-lift jump” would likely be selected as a candidate reward function. However, such candidates show limited improvement in subsequent iterations, failing to evolve into more human-like jumping behaviors.

Conversely, when real human preferences were used to guide the task, the results were notably different. The volunteer judged the overall quality of the humanoid’s jump behavior instead of just the metric of leaving the ground. Fig. 5 illustrates an example where the volunteer successfully guided the humanoid towards a more human-like jump by selecting behaviors that, while initially not optimal, displayed promising movement patterns. The reward functions are shown in Appendix A.6.3. In the first iteration, “leg-lift jump” was not selected despite the humanoid jumping off the ground. Instead, a video where the humanoid appears to attempt a jump using both legs, without leaving the ground, was chosen. By the fifth and sixth iterations, the humanoid demonstrated more sophisticated behaviors, such as bending both legs and lowering the upper body to shift the center of mass, behaviors that are much more akin to a real human jump.

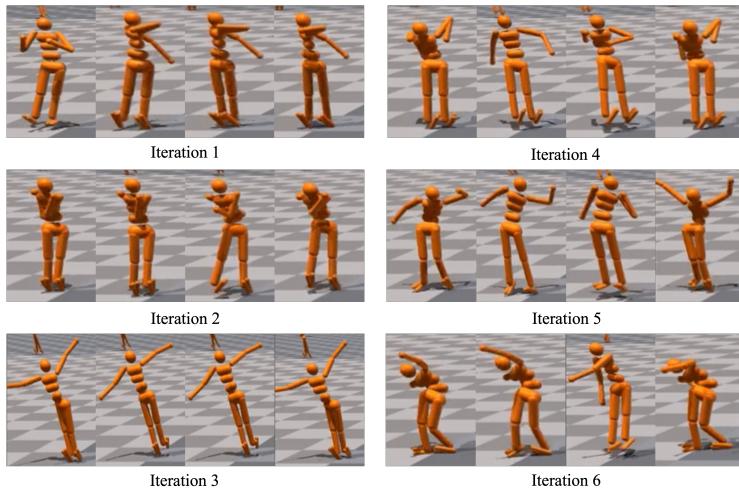


Figure 5: The humanoid learns a human-like jump by bending both legs and lowering the upper body to shift the center of mass in a trial of human-in-the-loop experiments. Note that both legs are used to jump and the agent bends at the hips.

**Quantitative Evaluation.** We conducted additional experiments using the “OpenLoop” configuration, which generates  $K \times N$  reward functions without any feedback, on the HumanoidJump task. In this configuration, we performed 5 independent experiments, each comprising 6 iterations with 6 samples per iteration. A volunteer selected the most preferred video as the final result. For quantitative evaluation, 20 additional volunteers were recruited to compare the performance of ICPL and OpenLoop. Each volunteer indicated their preference between two videos presented in random order—one generated by ICPL and the other by OpenLoop. The results showed that 17 out of 20 participants preferred the ICPL agent, demonstrating that ICPL produces behaviors more aligned with human preferences.

## 6 CONCLUSION

Our proposed method, In-Context Preference Learning (ICPL), demonstrates significant potential for addressing the challenges of preference learning tasks through the integration of large language models. By leveraging the generative capabilities of LLMs to autonomously produce reward functions, and iteratively refining them using human feedback, ICPL reduces the complexity and human effort typically associated with preference-based RL. Our experimental results, both in proxy human and human-in-the-loop settings, show that ICPL not only surpasses traditional RLHF in efficiency but also competes effectively with methods utilizing ground-truth rewards instead of preferences. Furthermore, the success of ICPL in complex, subjective tasks like humanoid jumping highlights its versatility in capturing nuanced human intentions, opening new possibilities for future applications in complex real-world scenarios where traditional reward functions are difficult to define.

**Limitations.** While ICPL demonstrates significant potential, it faces limitations in tasks where human evaluators struggle to assess performance from video alone, such as *Anymal*'s “follow random commands.” In such cases, subjective human preferences may not provide adequate guidance. Future work will explore integrating human preferences with artificially designed metrics to enhance the ease with which humans can assess the videos, ensuring more reliable performance in complex tasks. Additionally, we observe that the performance of the task is qualitatively dependent on the diversity of the initial reward functions that seed the search. While we do not study methods to achieve this here, relying on the LLM to provide this initial diversity is a current limitation. Furthermore, the limited number of participants in human-in-the-loop experiments may restrict the generalizability of our findings, as it might not fully capture the broad range of human preferences. Another limitation of ICPL is that each iteration involves training new RL policies, resulting in a waiting period of several hours for participants before they can provide additional feedback. This could be addressed by continuously training an RL agent under non-stationary reward functions, which presents a promising direction for future work.

Method	Vote
OpenLoop	3/20
ICPL	17/20

Table 4: Human Preferences

540 REFERENCES  
541

- 542 Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, meth-  
543 ods and progress. *Artificial Intelligence*, 297:103500, 2021.
- 544 Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the poten-  
545 tial of prompt engineering in large language models: a comprehensive review. *arXiv preprint*  
546 *arXiv:2310.14735*, 2023.
- 547 Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep  
548 reinforcement learning from human preferences. *Advances in neural information processing sys-  
549 tems*, 30, 2017.
- 550 Yuqing Du, Ksenia Konyushkova, Misha Denil, Akhil Raju, Jessica Landon, Felix Hill, Nando  
551 de Freitas, and Serkan Cabi. Vision-language models as success detectors. *arXiv preprint*  
552 *arXiv:2303.07280*, 2023.
- 553 Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang,  
554 De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied  
555 agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:  
556 18343–18362, 2022.
- 557 Louie Giray. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical  
558 engineering*, 51(12):2629–2633, 2023.
- 559 Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy  
560 maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL <https://arxiv.org/abs/1801.01290>.
- 561 Simon Holk, Daniel Marta, and Iolanda Leite. Predilect: Preferences delineated with zero-shot  
562 language-based reasoning in reinforcement learning. In *2024 19th ACM/IEEE International Con-  
563 ference on Human-Robot Interaction (HRI)*, pp. 259–268, 2024.
- 564 Borja Ibarz, Jan Leike, Tobias Pohlen, Geoffrey Irving, Shane Legg, and Dario Amodei. Reward  
565 learning from human preferences and demonstrations in atari. *Advances in neural information  
566 processing systems*, 31, 2018.
- 567 Hong Jun Jeon, Smitha Milli, and Anca Dragan. Reward-rational (implicit) choice: A unifying  
568 formalism for reward learning. *Advances in Neural Information Processing Systems*, 33:4415–  
569 4426, 2020.
- 570 Siddharth Karamcheti, Suraj Nair, Annie S Chen, Thomas Kollar, Chelsea Finn, Dorsa Sadigh,  
571 and Percy Liang. Language-driven representation learning for robotics. *arXiv preprint*  
572 *arXiv:2302.12766*, 2023.
- 573 Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language  
574 models. *arXiv preprint arXiv:2303.00001*, 2023.
- 575 Kimin Lee, Laura Smith, Anca Dragan, and Pieter Abbeel. B-pref: Benchmarking preference-based  
576 reinforcement learning. In *Thirty-fifth Conference on Neural Information Processing Systems  
577 Datasets and Benchmarks Track (Round 1)*.
- 578 Kimin Lee, Laura Smith, and Pieter Abbeel. Pebble: Feedback-efficient interactive rein-  
579 forcement learning via relabeling experience and unsupervised pre-training. *arXiv preprint*  
580 *arXiv:2106.05091*, 2021a.
- 581 Kimin Lee, Laura Smith, and Pieter Abbeel. Pebble: Feedback-efficient interactive reinforcement  
582 learning via relabeling experience and unsupervised pre-training, 2021b. URL <https://arxiv.org/abs/2106.05091>.
- 583 Kimin Lee, Laura Smith, Anca Dragan, and Pieter Abbeel. B-pref: Benchmarking preference-based  
584 reinforcement learning. In *Thirty-fifth Conference on Neural Information Processing Systems  
585 Datasets and Benchmarks Track (Round 1)*, 2021c. URL [https://openreview.net/forum?id=ps95-mkHF\\_](https://openreview.net/forum?id=ps95-mkHF_).

- 594 Fei Liu et al. Learning to summarize from human feedback. In *Proceedings of the 58th Annual*  
 595 *Meeting of the Association for Computational Linguistics*, 2020.
- 596
- 597 Yecheng Jason Ma, Shagun Sodhani, Dinesh Jayaraman, Osbert Bastani, Vikash Kumar, and Amy  
 598 Zhang. Vip: Towards universal visual reward and representation via value-implicit pre-training.  
 599 *arXiv preprint arXiv:2210.00030*, 2022.
- 600
- 601 Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayara-  
 602 man, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via  
 603 coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.
- 604
- 605 Yecheng Jason Ma, William Liang, Hung-Ju Wang, Sam Wang, Yuke Zhu, Linxi Fan, Osbert Bas-  
 606 tani, and Dinesh Jayaraman. Dreureka: Language model guided sim-to-real transfer. *arXiv*  
 607 *preprint arXiv:2406.01967*, 2024.
- 608
- 609 Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and  
 610 Ángel Fernández-Leal. Human-in-the-loop machine learning: a state of the art. *Artificial Intelli-  
 611 gence Review*, 56(4):3005–3054, 2023.
- 612
- 613 Soroush Nasiriany, Fei Xia, Wenhao Yu, Ted Xiao, Jacky Liang, Ishita Dasgupta, Annie Xie, Danny  
 614 Driess, Ayzaan Wahid, Zhuo Xu, et al. Pivot: Iterative visual prompting elicits actionable knowl-  
 615 edge for vlms. *arXiv preprint arXiv:2402.07872*, 2024.
- 616
- 617 Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, vol-  
 618 ume 1, pp. 2, 2000.
- 619
- 620 Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong  
 621 Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to fol-  
 622 low instructions with human feedback. *Advances in neural information processing systems*, 35:  
 623 27730–27744, 2022.
- 624
- 625 Jongjin Park, Younggyo Seo, Jinwoo Shin, Honglak Lee, Pieter Abbeel, and Kimin Lee. SURF:  
 626 Semi-supervised reward learning with data augmentation for feedback-efficient preference-based  
 627 reinforcement learning. In *International Conference on Learning Representations*, 2022. URL  
 628 <https://openreview.net/forum?id=TfhfZLQ2EJ0>.
- 629
- 630 Zhenghao Mark Peng, Wenjie Mo, Chenda Duan, Quanyi Li, and Bolei Zhou. Learning from active  
 631 human involvement through proxy value propagation. *Advances in neural information processing*  
 632 *systems*, 36, 2024.
- 633
- 634 Carl Orge Retzlaff, Srijita Das, Christabel Wayllace, Payam Mousavi, Mohammad Afshari, Tianpei  
 635 Yang, Anna Saranti, Alessa Angerschmid, Matthew E Taylor, and Andreas Holzinger. Human-in-  
 636 the-loop reinforcement learning: A survey and position on requirements, challenges, and oppor-  
 637 tunities. *Journal of Artificial Intelligence Research*, 79:359–415, 2024.
- 638
- 639 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy  
 640 optimization algorithms, 2017. URL <https://arxiv.org/abs/1707.06347>.
- 641
- 642 Satinder Singh, Richard L Lewis, and Andrew G Barto. Where do rewards come from. In *Proce-  
 643 dings of the annual conference of the cognitive science society*, pp. 2601–2606. Cognitive Science  
 644 Society, 2009.
- 645
- 646 Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.
- 647
- 648 Yufei Wang, Zhanyi Sun, Jesse Zhang, Zhou Xian, Erdem Biyik, David Held, and Zackory Erick-  
 649 son. RI-vlm-f: Reinforcement learning from vision language foundation model feedback. *arXiv*  
 650 *preprint arXiv:2402.03681*, 2024.
- 651
- 652 Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf El-  
 653 nashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance  
 654 prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

648 Christian Wirth, Riad Akroud, Gerhard Neumann, and Johannes Fürnkranz. A survey of preference-  
649 based reinforcement learning methods. *Journal of Machine Learning Research*, 18(136):1–46,  
650 2017.

651  
652 Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Chris-  
653 tiano. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*,  
654 2021.

655 Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Are-  
656 nas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to  
657 rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023a.

658 Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montserrat Gonzalez  
659 Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter,  
660 Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval  
661 Tassa, and Fei Xia. Language to rewards for robotic skill synthesis. In Jie Tan, Marc Toussaint,  
662 and Kourosh Darvish (eds.), *Conference on Robot Learning, Corl 2023, 6-9 November 2023,*  
663 *Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pp. 374–404.  
664 PMLR, 2023b. URL <https://proceedings.mlr.press/v229/yu23a.html>.

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702            **A APPENDIX**

703

704            We would suggest visiting <https://sites.google.com/view/few-shot-icpl/home> for more in-  
 705            formation and videos.

706

707            **A.1 FULL PROMPTS**

708

709            The prompts used in ICPL for synthesizing reward functions are presented in Prompts 1, 2, and 3.  
 710            The prompt for generating the differences between various reward functions is shown in Prompt 4.

711

712            **Prompt 1: Initial System Prompts of Synthesizing Reward Functions**

```
713 You are a reward engineer trying to write reward functions to solve reinforcement learning
714 tasks as effective as possible.
715 Your goal is to write a reward function for the environment that will help the agent learn the
716 task described in text.
717 Your reward function should use useful variables from the environment as inputs. As an example
718 , the reward function signature can be:
719 @torch.jit.script
720 def compute_reward(object_pos: torch.Tensor, goal_pos: torch.Tensor) -> Tuple[torch.Tensor,
721 Dict[str, torch.Tensor]]:
722 ...
723     return reward, {}
724 Since the reward function will be decorated with @torch.jit.script, please make sure that the
725 code is compatible with TorchScript (e.g., use torch tensor instead of numpy array).
726 Make sure any new tensor or variable you introduce is on the same device as the input tensors.
```

727            **Prompt 2: Feedback Prompts**

728

```
729 The reward function has been iterated {current_iteration} rounds.
730 In each iteration, a good reward function and a bad reward function are generated.
731 The good reward function generated in the x-th iteration is denoted as "iterx-good", and the
732 bad reward function generated is denoted as "iterx-bad".
733 The following outlines the differences between these reward functions.
734
735 We trained an RL policy using iter1-good reward function code and tracked the values of the
736 individual components in the reward function after every {epoch_freq} epochs and the
737 maximum, mean, minimum values encountered:
738 <REWARD FEEDBACK>
739
740 The difference between iter2-good and iter1-good is: <DIFFERENCE>
741
742 <REPEAT UNTIL THE CURRENT ITERATION>
743
744 Next, the two reward functions generated in the {current_iteration_ordinal} iteration are
745 provided.
746 The 1st generated reward function is as follows:
747 <REWARD FUNCTION>
748 We trained an RL policy using the 1st reward function code and tracked the values of the
749 individual components in the reward function after every {epoch_freq} epochs and the
750 maximum, mean, minimum values encountered:
751 <REWARD FEEDBACK>
752
753 The 2nd generated reward function is as follows:
754 <REWARD FUNCTION>
755 We trained an RL policy using the 2nd reward function code and tracked the values of the
756 individual components in the reward function after every {epoch_freq} epochs and the
757 maximum, mean, minimum values encountered:
758 <REWARD FEEDBACK>
759
760 The following content is the most important information.
761 Good example: 1st reward function. Bad example: 2nd reward function.
762 You need to modify based on the good example. DO NOT based on the code of the bad example.
763 Please carefully analyze the policy feedback and provide a new, improved reward function that
764 can better solve the task. Some helpful tips for analyzing the policy feedback:
765 (1) If the values for a certain reward component are near identical throughout, then this
766 means RL is not able to optimize this component as it is written. You may consider
767 (a) Changing its scale or the value of its temperature parameter
768 (b) Re-writing the reward component
769 (c) Discarding the reward component
770 (2) If some reward components' magnitude is significantly larger, then you must re-scale
771 its value to a proper range
772 Please analyze each existing reward component in the suggested manner above first, and then
773 write the reward function code.
```

756

### Prompt 3: Prompts of Tips for Writing Reward Functions

757

```
The output of the reward function should consist of two items:  

    (1) the total reward,  

    (2) a dictionary of each individual reward component.  

The code output should be formatted as a python code string: "```python ... ```".  

Some helpful tips for writing the reward function code:  

    (1) You may find it helpful to normalize the reward to a fixed range by applying  

        transformations like torch.exp to the overall reward or its components  

    (2) If you choose to transform a reward component, then you must also introduce a  

        temperature parameter inside the transformation function; this parameter must be a named  

        variable in the reward function and it must not be an input variable. Each transformed  

        reward component should have its own temperature variable  

    (3) Make sure the type of each input variable is correctly specified; a float input  

        variable should not be specified as torch.Tensor  

    (4) Most importantly, the reward code's input variables must contain only attributes of  

        the provided environment class definition (namely, variables that have prefix self.).  

        Under no circumstance can you introduce new input variables.
```

760

761

### Prompt 4: Prompts of Describing Differences

762

```
You are an engineer skilled at comparing the differences between two reward function code  

snippets used in reinforcement learning.  

Your goal is to describe the differences between two reward function code snippets.  

The following are two reward functions written in Python code used for the task:  

<TASK_DESCRIPTION>  

The first reward function is as follows:  

<REWARD_FUNCTION>  

The second reward function is as follows:  

<REWARD_FUNCTION>  

Please directly describe the differences between these two codes. No additional descriptions  

other than the differences are required.
```

779

780

## A.2 ICPL DETAILS

781

The full pseudocode of ICPL is listed in Algo. 2.

782

783

## A.3 BASELINE DETAILS

784

### A.3.1 PREFPPO

785

The baseline PrefPPO adopted in our experiments comprises two primary components: agent learning and reward learning, as outlined in Lee et al. (2021c). Algo. 3 illustrates the pseudocode for PrefPPO. Throughout this process, the method maintains a policy denoted as  $\pi_\phi$  and a reward model represented by  $r_\psi^*$ .

792

**Agent Learning.** In the agent learning phase, the agent interacts with the environment and collects experiences. The policy is subsequently trained using reinforcement learning, to maximize the cumulative rewards provided by the reward model  $r_\psi^*$ . We utilize the on-policy reinforcement learning algorithm PPO (Schulman et al., 2017) as the backbone algorithm for training the policy. Additionally, we apply unsupervised pre-training to match the performance of the original benchmark. Specifically, during earlier iterations, when the reward model has not collected sufficient trajectories and exhibits limited progress, we utilize the state entropy of the observations, defined as  $H(s) = -\mathbb{E}_{s \sim p(s)}[\log p(s)]$ , as the goal for agent training. During this process, trajectories of varying lengths are collected. Formally, a trajectory  $\sigma$  is defined as a sequence of observations and actions  $(s_1, a_1), \dots, (s_t, a_t)$  that represents the complete interaction of the agent with the environment, concluding at timestep  $t$ .

803

**Reward Learning.** A preference predictor is developed using the current reward model to align with human preferences, formulated as follows:

804

805

806

807

808

809

$$P_\psi[\sigma^1 \succ \sigma^0] = \frac{\exp(\sum_t \hat{r}_\psi(s_t^1, a_t^1))}{\sum_{i \in \{0,1\}} \exp(\sum_t \hat{r}_\psi(s_t^i, a_t^i))}, \quad (1)$$

where  $\sigma_0 = (s_1^0, a_1^0), \dots, (s_{l_0}^0, a_{l_0}^0)$  and  $\sigma_1 = (s_1^1, a_1^1), \dots, (s_{l_1}^1, a_{l_1}^1)$  represent two complete trajectories with different trajectory length  $l_0$  and  $l_1$ .  $P_\psi[\sigma^1 \succ \sigma^0]$  denotes the probability that trajectory

810  
811  
812  
813  
814  
815  
816  
817  
818  
819

---

**Algorithm 2:** ICPL

---

```

820
821
822 Input: # iterations  $N$ , # samples in each iterations  $K$ , environment Env, coding LLM  $LLM_{RF}$ ,
823           difference LLM  $LLM_{Diff}$ 
824 1 Function Feedback(Env, RF):
825   return The values of each component that make up RF during the training process in Env
826 2 Function History(RFlist, Env, LLMDiff):
827   HistoryFeedback  $\leftarrow$  ""
828   for  $i \leftarrow 1$  to  $\text{len}(RFlist) - 1$  do
829     // The reward trace of historical reward functions
830     HistoryFeedback  $\leftarrow$  HistoryFeedback + Feedback(Env, RFlist[ $i - 1$ ])
831     // The differences between historical reward functions
832     HistoryFeedback  $\leftarrow$ 
833       HistoryFeedback + LLMDiff(DifferencePrompt + RFlist[ $i$ ] + RFlist[ $i - 1$ ])
834   end
835   return HistoryFeedback
836   // Initialize the prompt containing the environment context and task description
837 10 Prompt  $\leftarrow$  InitializePrompt
838 11 RFlist  $\leftarrow$  []
839 12 for  $i \leftarrow 1$  to  $N$  do
840   RF1, ..., RF $K$   $\leftarrow$  LLMRF(Prompt,  $K$ )
841   while any of RF1, ..., RF $K$  is not executable do
842     j1, ..., j $K'$   $\leftarrow$  Index of non-executable reward functions
843     // Regenerate non-executable reward functions
844     RFj1, ..., RFj $K'$   $\leftarrow$  LLMRF(Prompt,  $K'$ )
845   end
846   // Render videos for sampled reward functions
847   Video1, ..., Video $K$   $\leftarrow$  Render(Env, RF1), ..., Render(Env, RF $K$ )
848   // Human selects the most preferred and least preferred videos
849   G, B  $\leftarrow$  Human(Video1, ..., Video $K$ )
850   GoodRF, BadRF  $\leftarrow$  RFG, RFB
851   RFlist.append(GoodRF)
852   // Update prompt for feedback
853   Prompt  $\leftarrow$ 
854     GoodRF + Feedback(Env, GoodRF) + BadRF + Feedback(Env, BadRF) + PreferencePrompt
855   Prompt  $\leftarrow$  Prompt + History(RFlist, Env, LLMDiff)
856 24 end
857
858
859
860
861
862
863
```

---

$\sigma^1$  is preferred over  $\sigma^0$  as indicated by the preference predictor. In the original PrefPPO framework, test task trajectories are of fixed length, allowing for the extraction of fixed-length segments to train the reward model. However, the tasks in this paper have varying trajectory lengths, so we use full trajectory pairs as training data instead of segments. We also tried zero-padding trajectories to the maximum episode length and then segmenting them, but this approach was ineffective in practice.

To provide more effective labels, the original PrefPPO utilizes dense rewards  $r$  to simulate oracle human preferences, which is

$$P[\sigma^1 \succ \sigma^0] = \begin{cases} 1 & \text{If } \sum_t r(s_t^1, a_t^1) > \sum_t r(s_t^0, a_t^0) \\ 0 & \text{Otherwise} \end{cases}. \quad (2)$$

The probability  $P[\sigma^1 \succ \sigma^0]$  reflects the preference of the ideal teacher, which is perfectly rational and deterministic, without incorporating noise. We utilize the default dense rewards in the adopted IsaacGym tasks, which differ from ICPL and EUREKA, both of which use sparse rewards (task metrics) as the proxy preference. While we also experimented with sparse rewards in PrefPPO and found similar performance (refer to Table 8), we opted to retain the original PrefPPO approach in all experiments. The reward model is trained by minimizing the cross-entropy loss between the predictor and labels, utilizing trajectories sampled from the agent learning process. Note that since the agent learning process requires significantly more experiences for training than reward training, we only use trajectories from a subset of the environments for reward training.

To sample trajectories for reward learning, we employ the disagreement sampling scheme from Lee et al. (2021c) to enhance the training process. This scheme first generates a larger batch of trajectory pairs uniformly at random and then selects a smaller batch with high variance across an ensemble of preference predictors. The selected pairs are used to update the reward model.

For a fair comparison, we recorded the number of times PrefPPO queried the oracle human simulator to compare two trajectories and obtain labels during the reward learning process, using this as a measure of the human effort involved. In the proxy human experiment, we set the maximum number of human queries  $Q$  to 49, 150, 1.5k, 15k. Once this limit is reached, the reward model ceases to update, and only the policy model is updated via PPO. Algo. 4 illustrates the pseudocode for reward learning.

### A.3.2 PEBBLE

PEBBLE (Lee et al., 2021b) is a popular feedback-efficient preference-based RL algorithm. It improves the feedback efficiency of the algorithm by mainly utilizing two modules: unsupervised pre-training and off-policy learning. The unsupervised pre-training module is introduced in the PrefPPO section, and we also include it in PEBBLE with the same setting. PEBBLE utilizes the off-policy algorithm SAC (Haarnoja et al., 2018) instead of PPO as the backbone RL algorithm. SAC stores the agent’s past experiences in a replay buffer and reuses these experiences during the training process. PEBBLE relabels all past experiences in the replay buffer every time it updates the reward model.

### A.3.3 SURF

SURF (Park et al., 2022) is a framework that uses unlabeled samples with data augmentation to improve the efficiency of reward training. In our experiments, the length of trajectories is varied and may affect the evaluation of the trajectories. Therefore, we do not apply the data augmentation technique and only utilize the semi-supervised learning method in SURF.

In addition to the labeled pairs of trajectories  $\mathcal{D}_l = \{(\sigma_l^0, \sigma_l^1, y)^i\}_{i=1}^{N_l}$ , SURF samples another unlabeled dataset  $\mathcal{D}_U = \{(\sigma_u^0, \sigma_u^1)^i\}_{u=1}^{N_u}$  to optimize the reward model. Specifically, during each update of the reward model, SURF not only samples a set of trajectories and queries a human teacher for labels, but also samples additional trajectory pairs. These additional pairs are assigned pseudo-labels generated by the current reward model.

$$\hat{y}_u(\sigma_u^0, \sigma_u^1) = \begin{cases} 1 & \text{If } P_\psi[\sigma_u^1 \succ \sigma_u^0] > 0.5. \\ 0 & \text{Otherwise.} \end{cases} \quad (3)$$

Here  $\psi$  is the preference predictor based on the current reward model. During the training process of reward model, SURF will also use the unlabeled samples for training if the confidence of the

918 predictor is higher than a pre-defined threshold. In experiments, we follows the implementation of  
 919 SURF (Park et al., 2022).  
 920

---

**Algorithm 3:** PrefPPO
 

---

```

921 Input: # iterations  $B$ , # unsupervised learning iterations  $M$ , # rollout steps  $S$ , reward model
922    $\hat{r}_\psi$ , # environments for reward learning  $E$ , # iterations for collecting trajectories
923   RewardTrainingInterval, maximal number of human queries  $Q$ , environments Env
924
925 1 HumanQueryCount  $\leftarrow 0$ 
926 2 Trajectories  $\leftarrow []$ 
927 3 Function TrainReward( $\hat{r}_\psi$ , Trajectories):
928 4 Function CollectRollout(RewardType,  $S$ , Policy,  $\hat{r}_\psi$ , Env):
929   RolloutBuffer  $\leftarrow []$ 
930   for  $j \leftarrow 1$  to  $S$  do
931     Action  $\leftarrow$  Policy(Observation)
932     // Here EnvDones is a binary sequence replied from the environment,
933     // representing whether the environments are done.
934     NewObservation, EnvReward, EnvDones  $\leftarrow$  Env(Actions)
935     if RewardType == Unsuper then
936       | PredReward  $\leftarrow$  ComputeStateEntropy(Observation)
937     end
938     else
939       | PredReward  $\leftarrow \hat{r}_\psi(\text{Observation}, \text{Action})$ 
940     end
941     // Collect trajectories for reward learning
942     Trajectories  $\leftarrow$  Trajectories + (Observation, Action, EnvDones, EnvReward)
943     // Add complete trajectory to reward model
944     for  $k \leftarrow 1$  to  $E$  do
945       | if EnvDones[Env[k]] then
946         | | AddTrajectory( $\hat{r}_\psi$ , Trajectories[k])
947         | | Trajectories[k]  $\leftarrow []$ 
948       end
949     end
950     // Reward Learning
951     if  $j$  is divisible by RewardTrainingInterval and HumanQueryCount  $< Q$  then
952       | |  $\hat{r}_\psi \leftarrow$  TrainReward( $\hat{r}_\psi$ , Trajectories)
953     end
954     // Collect rollouts for agent learning
955     RolloutBuffer  $\leftarrow$  RolloutBuffer + (Observation, Action, PredReward)
956     Observation  $\leftarrow$  NewObservation
957   end
958   return RolloutBuffer
959   Policy  $\leftarrow$  Initialize
960   for  $i \leftarrow 1$  to  $B$  do
961     // Collect rollouts and trajectories
962     if  $i < M$  then
963       | | RolloutBuffer  $\leftarrow$  CollectRollout(Unsuper,  $S$ , Policy,  $\hat{r}_\psi$ , Env)
964     end
965     else
966       | | RolloutBuffer  $\leftarrow$  CollectRollout(RewardModel,  $S$ , Policy,  $\hat{r}_\psi$ , Env)
967     end
968     // Agent Learning: Train agent with the collect RolloutBuffer via PPO, omitted
969     // here
970     AgentLearning(Policy, RolloutBuffer)
971   end

```

---

**Algorithm 4:** Reward Learning of PrefPPO

---

**Input:** reward model  $\hat{r}_\psi$ , # samples for human queries per time MbSize, # maximal iterations for reward learning MaxUpdate, maximal number of human queries  $Q$ , environments Env

```

1 LabeledQueries← []
2 HumanQueryCount← 0
3 Function TrainReward( $\hat{r}_\psi$ , Trajectories):
4     // Use disagreement sampling to sample trajectories
5      $\sigma_0, \sigma_1 \leftarrow$  DisagreementSampling(Trajectories, MbSize)
6     for  $(x_0, x_1)$  in  $(\sigma_0, \sigma_1)$  do
7         // Give oracle human preferences between two trajectories according to the sum
8         // of dense reward.
9         LabeledQueries← LabeledQueries +  $(x_0, x_1, \text{HumanQuery}(x_0, x_1))$ 
10        // In experiments, we do not add HumanQueryCount if the pair has already been
11        // queried before
12        HumanQueryCount ← HumanQueryCount + 1
13        if HumanQueryCount >  $Q$  then
14            | BREAK
15        end
16    end
17    for  $i \leftarrow 1$  to MaxUpdate do
18        // Update reward model by minimizing the cross entropy loss and record the
19        // accuracy on all pairs.
20         $\hat{r}_\psi, \text{Accuracy} \leftarrow$  RewardLearning( $\hat{r}_\psi$ , LabeledQueries)
21        if Accuracy  $\geq 97\%$  then
22            | BREAK
23        end
24    end
25    return  $\hat{r}_\psi$ 

```

## A.4 ENVIRONMENT DETAILS

In Table 5, we present the observation and action dimensions, along with the task description and task metrics for 9 tasks in IsaacGym.

## A.5 PROXY HUMAN PREFERENCE

### A.5.1 ADDITIONAL RESULTS

Due to the high variance in LLMs performance, we report the standard deviation across 5 experiments as a supplement, which is presented in Table 6 and Table 7. We also report the final task score of PrefPPO using sparse rewards as the preference metric for the simulated teacher in Table 8.

### A.5.2 IMPROVEMENT ANALYSIS

We use a trial of the *Humanoid* task to illustrate how ICPL progressively generated improved reward functions over successive iterations. The task description is “to make the humanoid run as fast as possible”. Throughout five iterations, adjustments were made to the penalty terms and reward weightings. In the first iteration, the total reward was calculated as  $0.5 \times \text{speed\_reward} + 0.25 \times \text{deviation\_reward} + 0.25 \times \text{action\_reward}$ , yielding an RTS of 5.803. The speed reward and deviation reward motivate the humanoid to run fast, while the action reward promotes smoother motion. In the second iteration, the weight of the speed reward was increased to 0.6, while the weights for deviation and action rewards were adjusted to 0.2 each, improving the RTS to 6.113. In the third iteration, the action penalty was raised and the reward weights were further modified to  $0.7 \times \text{speed\_reward}$ ,  $0.15 \times \text{deviation\_reward}$ , and  $0.15 \times \text{action\_reward}$ , resulting in an RTS of 7.915. During the fourth iteration, the deviation penalty was reduced to 0.35 and the action penalty was lowered, with the reward weights set to 0.8, 0.1, and 0.1 for speed, deviation, and action rewards, respectively. This

1026	<b>Environment (obs dim, action dim)</b>
1027	Task Description
1028	<i>Task Metric</i>
1029	
1030	<b>Cartpole (4, 1)</b>
1031	To balance a pole on a cart so that the pole stays upright <i>duration</i>
1032	
1033	<b>Quadcopter (21, 12)</b>
1034	To make the quadcopter reach and hover near a fixed position <i>-cur_dist</i>
1035	
1036	<b>FrankaCabinet (23, 9)</b>
1037	To open the cabinet door
1038	<i>1 if cabinet_pos &gt; 0.39</i>
1039	
1040	<b>Anymal (48, 12)</b>
1041	To make the quadruped follow randomly chosen x, y, and yaw target velocities <i>-(linvel_error + angvel_error)</i>
1042	
1043	<b>BallBalance (48, 12)</b>
1044	To keep the ball on the table top without falling <i>duration</i>
1045	
1046	<b>Ant (60, 8)</b>
1047	To make the ant run forward as fast as possible <i>cur_dist - prev_dist</i>
1048	
1049	<b>AllegroHand (88, 16)</b>
1050	To make the hand spin the object to a target orientation <i>number of consecutive successes where current success is 1 if rot_dist &lt; 0.1</i>
1051	
1052	<b>Humanoid (108, 21)</b>
1053	To make the humanoid run as fast as possible <i>cur_dist - prev_dist</i>
1054	
1055	<b>ShadowHand (211, 20)</b>
1056	To make the shadow hand spin the object to a target orientation <i>number of consecutive successes where current success is 1 if rot_dist &lt; 0.1</i>

Table 5: Details of IssacGym Tasks.

	Cart.	Ball.	Quad.	Anymal	Ant	Human.	Franka	Shadow	Allegro
PrefPPO-49	499(0)	499(0)	-1.066(0.16)	-1.861(0.03)	0.743(0.20)	0.457(0.09)	0.0044(0.00)	0.0746(0.02)	0.0125(0.003)
PEBBLE-49	499(0)	499(0)	-1.1904(0.14)	-1.521	5.9891	0.903	0.0453	0.2142	0.1467
SURF-49	499(0)	499(0)	-1.208(0.03)	-1.35	0.815	1.675	0.0039	0.15	0.1116
PrefPPO-150	499(0)	499(0)	-0.959(0.15)	-1.818(0.07)	0.171(0.05)	0.607(0.02)	0.0179(0.01)	0.0617(0.01)	0.0153(0.004)
PEBBLE-150	499(0)	499(0)	-1.059(0.07)	-1.436	7.257	3.254	0.0532	0.2369	0.2811
SURF-150	499(0)	499(0)	-1.114(0.06)	-1.42	4.246	4.312	0.0453	0.2096	0.2
PrefPPO-1.5k	499(0)	499(0)	-0.486(0.11)	-1.417(0.21)	4.458(1.30)	1.329(0.33)	0.3248(0.12)	0.0488(0.01)	0.0284(0.005)
PEBBLE-1.5k	499(0)	499(0)	-0.529(0.14)	-1.332	8.282	4.075	0.1622	0.2416	0.2615
SURF-1.5k	499(0)	499(0)	-0.308(0.06)	-1.278	7.921	2.999	0.2639	0.2355	0.2283
PrefPPO-15k	499(0)	499(0)	-0.250(0.06)	-1.357(0.02)	4.626(0.57)	1.317(0.34)	0.0399(0.02)	0.0468(0.00)	0.0157(0.003)
PEBBLE-15k	499(0)	499(0)	-0.231(0.04)	-0.730	8.543	4.074	0.6089	0.2438	0.2401
SURF-15k	499(0)	499(0)	-0.266(0.02)	-0.760	7.859	3.2922	0.3434	0.2145	0.2352
ICPL(Ours)	499(0)	499(0)	<b>-0.0195(0.09)</b>	<b>-0.007(0.35)</b>	<b>12.04(1.69)</b>	<b>9.227(0.93)</b>	<b>0.9999(0.24)</b>	<b>13.231(1.88)</b>	<b>25.030(3.721)</b>
Eureka	499(0)	499(0)	-0.023(0.07)	-0.003(0.38)	10.86(0.85)	9.059(0.83)	0.9999(0.23)	11.532(1.38)	25.250(9.583)

Table 6: The final task score of all methods across different tasks in IssacGym. The values in parentheses represent the standard deviation.

	Cart.	Ball.	Quad.	Anymal	Ant	Human.	Franka	Shadow	Allegro
ICPL w/o RT	499(0)	499(0)	-0.0340(0.05)	-0.387(0.26)	10.50(0.45)	8.337(0.60)	0.9999(0.25)	10.769(2.30)	25.641(9.46)
ICPL w/o RTD	499(0)	499(0)	-0.0216(0.14)	-0.009(0.38)	10.53(0.39)	9.419(2.10)	1.0000(0.18)	11.633(1.25)	23.744(8.80)
ICPL w/o RTDB	499(0)	499(0)	-0.0136(0.03)	-0.014(0.42)	11.97(0.71)	8.214(2.88)	0.5129(0.06)	13.663(1.83)	25.386(3.42)
OpenLoop	499(0)	499(0)	-0.0410(0.32)	-0.016(0.50)	9.350(2.34)	8.306(1.63)	0.9999(0.22)	9.476(2.44)	23.876(7.91)
ICPL(Ours)	499(0)	499(0)	-0.0195(0.09)	-0.007(0.35)	12.04(1.69)	9.227(0.93)	0.9999(0.24)	13.231(1.88)	25.030(3.721)

Table 7: Ablation studies on ICPL modules. The values in parentheses represent the standard deviation.

		Cart.	Ball.	Quad.	Anymal	Ant	Human.	Franka	Shadow	Allegro
1081	PrefPPO-49	499(0)	499(0)	-1.288(0.04)	-1.833(0.05)	0.281(0.06)	0.855(0.24)	0.0009(0.00)	0.1178(0.03)	0.1000(0.024)
1082	PrefPPO-150	499(0)	499(0)	-1.288(0.02)	-1.814(0.07)	0.545(0.16)	0.546(0.09)	0.0012(0.00)	0.0517(0.01)	0.0544(0.010)
1083	PrefPPO-1.5k	499(0)	499(0)	-1.292(0.05)	-1.583(0.13)	2.235(0.63)	2.480(0.59)	0.0077(0.00)	0.0495(0.01)	0.0667(0.017)
1084	PrefPPO-15k	499(0)	499(0)	-1.322(0.04)	-1.611(0.12)	3.694(0.86)	1.867(0.19)	0.0066(0.00)	0.0543(0.01)	0.1002(0.030)
	Eureka	499(0)	499(0)	-0.023(0.07)	-0.003(0.38)	10.86(0.85)	9.059(0.83)	0.9999(0.23)	11.532(1.38)	25.250(9.583)
	(Ours)	499(0)	499(0)	-0.0195(0.09)	-0.007(0.35)	12.04(1.69)	9.227(0.93)	0.9999(0.24)	13.231(1.88)	25.030(3.721)

Table 8: The final task score of all methods across different tasks in IssacGym, where PrefPPO uses sparse rewards as the preference metric for the simulated teacher. The values in parentheses represent the standard deviation.

change led to an RTS of 8.125. Finally, in the fifth iteration, an additional upright reward term was incorporated, with the total reward calculated as  $0.7 \times \text{speed\_reward} + 0.1 \times \text{deviation\_reward} + 0.1 \times \text{action\_reward} + 0.1 \times \text{upright\_reward}$ . This adjustment produced the highest RTS of 8.232, allowing ICPL to generate reward functions that were more effectively aligned with the task description. Below are the specific reward functions produced at each iteration during one experiment.

#### Humanoid Task: Reward Function with highest RTS (5.803) of Iteration 1

```

1098 def compute_reward(root_states: torch.Tensor, actions: torch.Tensor
1099 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1100     velocity = root_states[:, 7:10]
1101     forward_velocity = velocity[:, 0]
1102     target_velocity = 5.0
1103     deviation_penalty = 0.5
1104     action_penalty = 0.1
1105
1106     # Measure how fast the humanoid is going
1107     speed_reward = torch.exp((forward_velocity - target_velocity))
1108
1109     # Penalize deviation from the target velocity
1110     deviation = torch.abs(forward_velocity - target_velocity)
1111     deviation_reward = torch.exp(-deviation_penalty * deviation)
1112
1113     # Penalize actions, to induce smoother motion
1114     sum_actions = torch.sum(actions ** 2, dim=1)
1115     action_reward = torch.exp(-action_penalty * sum_actions)
1116
1117     rewards = {'speed_reward': speed_reward, 'deviation_reward':
1118               deviation_reward, 'action_reward': action_reward }
1119
1120     total_reward = 0.5 * speed_reward + 0.25 * deviation_reward +
1121     0.25 * action_reward
1122     total_reward = total_reward.to(deviation.device)
1123
1124     return total_reward, rewards

```

#### Humanoid Task: Reward Function with highest RTS (6.113) of Iteration 2

```

1125 def compute_reward(root_states: torch.Tensor, actions: torch.Tensor
1126 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1127     velocity = root_states[:, 7:10]
1128     forward_velocity = velocity[:, 0]
1129     target_velocity = 5.0
1130     deviation_penalty = 0.5
1131     action_penalty = 0.1
1132
1133     # Measure how fast the humanoid is going
1134     speed_reward = torch.exp((forward_velocity - target_velocity))

```

```

1134
1135     # Penalize deviation from the target velocity
1136     deviation = torch.abs(forward_velocity - target_velocity)
1137     deviation_reward = torch.exp(-deviation_penalty * deviation)
1138     # Penalize actions, to induce smoother motion
1139     sum_actions = torch.sum(actions ** 2, dim=1)
1140     action_reward = torch.exp(-action_penalty * sum_actions)
1141
1142     rewards = {'speed_reward': speed_reward, 'deviation_reward':
1143     deviation_reward, 'action_reward': action_reward }
1144
1145     # Adjusted weights in total reward calculation
1146     total_reward = 0.6 * speed_reward + 0.2 * deviation_reward +
1147     0.2 * action_reward # Changed weights
1148     total_reward = total_reward.to(deviation.device)
1149
1150
1151     return total_reward, rewards

```

## Humanoid Task: Reward Function with highest RTS (7.915) of Iteration 3

```

1153 def compute_reward(root_states: torch.Tensor, actions: torch.Tensor
1154 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1155     velocity = root_states[:, 7:10]
1156     forward_velocity = velocity[:, 0]
1157     target_velocity = 5.0
1158     deviation_penalty = 0.5
1159     action_penalty = 0.15 # Increased action penalty
1160
1161     # Measure how fast the humanoid is going
1162     speed_reward = torch.exp((forward_velocity - target_velocity))
1163     # Penalize deviation from the target velocity
1164     deviation = torch.abs(forward_velocity - target_velocity)
1165     deviation_reward = torch.exp(-deviation_penalty * deviation)
1166     # Penalize actions, to induce smoother motion
1167     sum_actions = torch.sum(actions ** 2, dim=1)
1168     action_reward = torch.exp(-action_penalty * sum_actions)
1169
1170     rewards = {'speed_reward': speed_reward, 'deviation_reward':
1171     deviation_reward, 'action_reward': action_reward }
1172
1173     # Adjusted weights in total reward calculation
1174     total_reward = 0.7 * speed_reward + 0.15 * deviation_reward +
1175     0.15 * action_reward # Changed weights
1176     total_reward = total_reward.to(deviation.device)
1177
1178     return total_reward, rewards

```

## Humanoid Task: Reward Function with highest RTS (8.125) of Iteration 4

```

1179 def compute_reward(root_states: torch.Tensor, actions: torch.Tensor
1180 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1181     velocity = root_states[:, 7:10]
1182     forward_velocity = velocity[:, 0]
1183     target_velocity = 5.0
1184     deviation_penalty = 0.35 # decrease penalty for deviation
1185     action_penalty = 0.1 # decrease penalty for action
1186
1187     # Measure how fast the humanoid is going
1188     speed_reward = torch.exp((forward_velocity - target_velocity))

```

```

1188
1189     # Penalize deviation from the target velocity
1190     deviation = torch.abs(forward_velocity - target_velocity)
1191     deviation_reward = torch.exp(-deviation_penalty * deviation)
1192     # Penalize actions, to induce smoother motion
1193     sum_actions = torch.sum(actions ** 2, dim=1)
1194     action_reward = torch.exp(-action_penalty * sum_actions)
1195
1196     rewards = {'speed_reward': speed_reward, 'deviation_reward':
1197     deviation_reward, 'action_reward': action_reward }
1198
1199     # Adjust the weights in total reward calculation
1200     total_reward = 0.8 * speed_reward + 0.1 * deviation_reward +
1201     0.1 * action_reward # Changed weights
1202     total_reward = total_reward.to(deviation.device)
1203
1204
1205     return total_reward, rewards
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241

```

#### Humanoid Task: Reward Function with highest RTS (8.233) of Iteration 5

```

1207 def compute_reward(root_states: torch.Tensor, actions: torch.Tensor,
1208 , up_vec: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.
1209 Tensor]]:
1210     velocity = root_states[:, 7:10]
1211     forward_velocity = velocity[:, 0]
1212     target_velocity = 5.0
1213     deviation_penalty = 0.35 # Reduced deviation penalty
1214     action_penalty = 0.1 # Decreased action penalty
1215
1216     # Measure how fast the humanoid is going
1217     speed_reward = torch.exp((forward_velocity - target_velocity))
1218     # Penalize deviation from the target velocity
1219     deviation = torch.abs(forward_velocity - target_velocity)
1220     deviation_reward = torch.exp(-deviation_penalty * deviation)
1221     # Penalize actions, to induce smoother motion
1222     sum_actions = torch.sum(actions ** 2, dim=1)
1223     action_reward = torch.exp(-action_penalty * sum_actions)
1224     # Reward for maintaining an upright position
1225     upright_penalty = 1.0 # New upright penalty for the humanoid
1226     upright_reward = torch.exp(-upright_penalty * (1 - up_vec[:, 2])) # Added upright reward
1227
1228     rewards = {'speed_reward': speed_reward, 'deviation_reward':
1229     deviation_reward, 'action_reward': action_reward, 'upright_reward': upright_reward }
1230
1231     # Adjusted weights in total reward calculation
1232     total_reward = 0.7 * speed_reward + 0.1 * deviation_reward +
1233     0.1 * action_reward + 0.1 * upright_reward # Added upright
1234     reward to total
1235     total_reward = total_reward.to(deviation.device)
1236
1237
1238
1239
1240
1241

```

## A.6 HUMAN-IN-THE-LOOP PREFERENCE

### A.6.1 DEMOGRAPHIC DATA

The participants in the human-in-the-loop preference experiments consisted of 7 individuals aged 19 to 30, including 2 women and 5 men. Their educational backgrounds included 2 undergraduate

1242 students and 5 graduate students. The 20 volunteers recruited to evaluate the performance of different methods were aged 23 to 28, comprising 5 women and 15 men, with 3 undergraduates and 17 graduate students.

#### 1246 A.6.2 ISAACGYM TASKS

1248 We evaluate human-in-the-loop preference experiments on tasks in IsaacGym, including *Quad-*  
 1249 *copter*, *Humanoid*, *Ant*, *ShadowHand*, and *AllegroHand*. In these experiments, volunteers were  
 1250 limited to comparing reward functions based solely on videos showcasing the final policies derived  
 1251 from each reward function.

1252 In the *Quadcopter* task, humans evaluate performance by observing whether the quadcopter moves  
 1253 quickly and efficiently, and whether it stabilizes in the final position. For the *Humanoid* and *Ant*  
 1254 tasks, where the task description is "make the ant/humanoid run as fast as possible," humans esti-  
 1255 mate speed by comparing the time taken to cover the same distance and assessing the movement  
 1256 posture. However, due to the variability in movement postures and directions, estimating speed can  
 1257 introduce inaccuracies. In the *ShadowHand* and *AllegroHand* tasks, where the goal is "to make  
 1258 the hand spin the object to a target orientation," Humans find it challenging to calculate the precise  
 1259 difference between the current orientation and the target orientation at every moment, even though  
 1260 the target orientation is displayed nearby. Nevertheless, humans still can estimate the duration of ef-  
 1261 fective rotations with the target orientation in the video, thus evaluating the performance of a single  
 1262 spin. Since the target orientation regenerates upon being reached, the frequency of target orientation  
 1263 changes can also aid in facilitating the assessment of evaluating performance.

1264 Due to the lack of precise environmental data, volunteers cannot make absolutely accurate judgments  
 1265 during the experiments. For instance, in the *Humanoid* task, robots may move in varying directions,  
 1266 which can introduce biases in volunteers' assessments of speed. However, volunteers are still able  
 1267 to filter out extremely poor results and select videos with relatively better performance. In most  
 1268 cases, the selected results closely align with those derived from proxy human preferences, enabling  
 1269 effective improvements in task performance.

1270 Below is a specific case from the *Humanoid* task that illustrates the potential errors humans  
 1271 may make during evaluation and the learning process of the reward function under this as-  
 1272 sumption. The reward task scores (RTS) chosen by the volunteer across five iterations are  
 1273 4.521, 6.069, 6.814, 6.363, 6.983.

1274 In the first iteration, the ground-truth task scores of each policy were  
 1275 0.593, 2.744, 4.520, 0.192, 2.517, 5.937, although the volunteer was unaware of these scores.  
 1276 Initially, the volunteer eliminated policies 0 and 3, as the robots in those videos primarily exhibited  
 1277 spinning behavior. Subsequently, the volunteer assessed the speed of the remaining robots based  
 1278 on how quickly a specific robot moved out of the field. The volunteer correctly identified that the  
 1279 robots in policies 1 and 4 were slightly slower. However, due to minor differences in the movement  
 1280 directions of the robots in policies 2 and 5, the volunteer mistakenly selected policy 2 as the best  
 1281 option, incorrectly concluding that the robot in policy 2 was faster.

1282 Thus, the reward function selected in iteration 1 consists of several key components: velocity reward,  
 1283 upright reward, force penalty, unnatural pose penalty, and action penalty. These components not only  
 1284 promote faster training, which is the primary objective, but also encourage the maintenance of an  
 1285 upright pose. Additionally, the function penalizes excessive force usage, extreme joint angles, and  
 1286 large action values to foster smoother and more controlled movements.

1287 In subsequent iterations, the volunteer effectively identified reward functions that exhibited relatively  
 1288 better and worse performance outcomes. Adjustments were made to the weights of each component,  
 1289 and specific temperature values were introduced for each. These modifications resulted in a more  
 1290 balanced reward structure, ensuring that critical aspects exert a stronger influence, thereby allowing  
 1291 for greater control over the learning dynamics and improving the agent's performance in achieving  
 1292 the task. Even in Iteration 4, the volunteer did not select the reward function with the highest RTS  
 1293 (6.813) but instead opted for the second-highest reward function (RTS = 6.363). Nevertheless, the  
 1294 reward function exhibited consistent improvement during these iterations.

1295 Here we show the full reward function during the process.

```

1296
1297 Humanoid Task: Reward Function chosen by volunteer with RTS (4.521) of Iteration 1
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
def compute_reward(
    velocity: torch.Tensor,
    dof_pos: torch.Tensor,
    dof_force_tensor: torch.Tensor,
    actions: torch.Tensor,
    up_vec: torch.Tensor,
    heading_vec: torch.Tensor
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Reward for forward velocity (run as fast as possible)
    velocity_reward = velocity[:, 0]

    # Encouragement for upright pose (penalize for deviation from vertical)
    upright_reward = up_vec[:, 2]

    # Penalize high force usage (energy efficiency)
    force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)

    # Penalize unnatural joint positions (for instance, avoid extreme angles)
    unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)

    # Penalize for large actions (to promote smoother movement)
    action_penalty = torch.sum(torch.abs(actions), dim=1)

    # Normalize the rewards and penalties
    velocity_reward = torch.exp(velocity_reward) - 1
    upright_reward = torch.exp(upright_reward) - 1
    temperature = 1.0
    force_penalty = torch.exp(-force_penalty / temperature)
    unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty / temperature)
    action_penalty = torch.exp(-action_penalty / temperature)

    # Combine the rewards and penalties into a single reward
    total_reward = (
        velocity_reward +
        0.5 * upright_reward -
        0.01 * force_penalty -
        0.01 * unnatural_pose_penalty -
        0.01 * action_penalty
    )

    # Return the total reward and each component for analysis
    reward_components = {
        "velocity_reward": velocity_reward,
        "upright_reward": upright_reward,
        "force_penalty": force_penalty,
        "unnatural_pose_penalty": unnatural_pose_penalty,
        "action_penalty": action_penalty
    }

    return total_reward, reward_components

```

Humanoid Task: Reward Function chosen by volunteer with RTS (6.069) of Iteration 2

```

def compute_reward(
    velocity: torch.Tensor,

```

```
1350
1351     dof_pos: torch.Tensor,
1352     dof_force_tensor: torch.Tensor,
1353     actions: torch.Tensor,
1354     up_vec: torch.Tensor,
1355     heading_vec: torch.Tensor
1356 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1357     # Reward for forward velocity (run as fast as possible)
1358     velocity_reward = velocity[:, 0]
1359     velocity_temperature = 1.2 # increased slightly
1360     velocity_reward = torch.exp(velocity_reward /
1361     velocity_temperature) - 1
1362
1363     # Encouragement for upright pose (penalize for deviation from
1364     # vertical)
1365     upright_reward = up_vec[:, 2]
1366     upright_temperature = 0.5 # introduce a specific temperature
1367     upright_reward = torch.exp(upright_reward / upright_temperature
1368 ) - 1
1369
1370     # Penalize high force usage (energy efficiency)
1371     force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)
1372     force_temperature = 0.1 # decreased to make it more
1373     significant
1374     force_penalty = torch.exp(-force_penalty / force_temperature)
1375
1376     # Penalize unnatural joint positions (for instance, avoid
1377     # extreme angles)
1378     unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)
1379     pose_temperature = 0.1 # decreased to make it more significant
1380     unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty /
1381     pose_temperature)
1382
1383     # Penalize for large actions (to promote smoother movement)
1384     action_penalty = torch.sum(torch.abs(actions), dim=1)
1385     action_temperature = 0.1 # decreased to make it more
1386     significant
1387     action_penalty = torch.exp(-action_penalty / action_temperature
1388 )
1389
1390     # Combine the rewards and penalties into a single reward
1391     total_reward = (
1392         velocity_reward +
1393         0.5 * upright_reward -
1394         0.01 * force_penalty -
1395         0.01 * unnatural_pose_penalty -
1396         0.01 * action_penalty
1397     )
1398
1399     # Return the total reward and each component for analysis
1400     reward_components = {
1401         "velocity_reward": velocity_reward,
1402         "upright_reward": upright_reward,
1403         "force_penalty": force_penalty,
1404         "unnatural_pose_penalty": unnatural_pose_penalty,
1405         "action_penalty": action_penalty
1406     }
1407
1408     return total_reward, reward_components
1409
```

Humanoid Task: Reward Function chosen by volunteer with RTS (6.814) of Iteration 3

```

1404
1405
1406
1407     def compute_reward(
1408         velocity: torch.Tensor,
1409         dof_pos: torch.Tensor,
1410         dof_force_tensor: torch.Tensor,
1411         actions: torch.Tensor,
1412         up_vec: torch.Tensor,
1413         heading_vec: torch.Tensor
1414     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1415         # Reward for forward velocity (run as fast as possible)
1416         velocity_reward = velocity[:, 0]
1417         velocity_temperature = 1.1 # minor adjustment
1418         velocity_reward = torch.exp(velocity_reward /
1419             velocity_temperature) - 1
1420
1421         # Encouragement for upright pose (penalize for deviation from
1422         # vertical)
1423         upright_reward = up_vec[:, 2]
1424         upright_temperature = 0.6 # slight adjustment
1425         upright_reward = torch.exp(upright_reward / upright_temperature
1426             ) - 1
1427
1428         # Penalize high force usage (energy efficiency)
1429         force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)
1430         force_temperature = 0.15 # increased to try to make it
1431         # effective
1432         force_penalty = torch.exp(-force_penalty / force_temperature)
1433
1434         # Penalize unnatural joint positions (for instance, avoid
1435         # extreme angles)
1436         unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)
1437         pose_temperature = 0.2 # increased to try to make it effective
1438         unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty /
1439             pose_temperature)
1440
1441         # Penalize for large actions (to promote smoother movement)
1442         action_penalty = torch.sum(torch.abs(actions), dim=1)
1443         action_temperature = 0.2 # increased to try to make it
1444         # effective
1445         action_penalty = torch.exp(-action_penalty / action_temperature
1446             )
1447
1448         # Combine the rewards and penalties into a single reward
1449         total_reward = (
1450             velocity_reward +
1451             0.5 * upright_reward -
1452             0.02 * force_penalty - # increased slightly for more
1453             impact
1454             0.02 * unnatural_pose_penalty - # increased slightly for
1455             more impact
1456             0.02 * action_penalty # increased slightly for more impact
1457         )
1458
1459         # Return the total reward and each component for analysis
1460         reward_components = {
1461             "velocity_reward": velocity_reward,
1462             "upright_reward": upright_reward,
1463             "force_penalty": force_penalty,
1464             "unnatural_pose_penalty": unnatural_pose_penalty,
1465             "action_penalty": action_penalty
1466         }
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
```

```

1458
1459     return total_reward, reward_components
1460
1461

```

Humanoid Task: Reward Function chosen by volunteer with RTS (6.363) of Iteration 4

```

1462
1463
1464
1465     def compute_reward(
1466         velocity: torch.Tensor,
1467         dof_pos: torch.Tensor,
1468         dof_force_tensor: torch.Tensor,
1469         actions: torch.Tensor,
1470         up_vec: torch.Tensor,
1471         heading_vec: torch.Tensor
1472     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1473         # Reward for forward velocity (run as fast as possible)
1474         velocity_reward = velocity[:, 0]
1475         velocity_temperature = 1.05 # slight adjustment to refine the
1476         impact
1477         velocity_reward = torch.exp(velocity_reward /
1478                                     velocity_temperature) - 1
1479
1480         # Encouragement for upright pose (penalize for deviation from
1481         # vertical)
1482         upright_reward = up_vec[:, 2]
1483         upright_temperature = 0.65 # slight loosening for more upright
1484         reward
1485         upright_reward = torch.exp(upright_reward / upright_temperature
1486                                     ) - 1
1487
1488         # Penalize high force usage (energy efficiency)
1489         force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)
1490         force_temperature = 0.2 # increased to make it more
1491         significant
1492         force_penalty = torch.exp(-force_penalty / force_temperature)
1493
1494         # Penalize unnatural joint positions (for instance, avoid
1495         # extreme angles)
1496         unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)
1497         pose_temperature = 0.25 # slight increase to make this
1498         component effective
1499         unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty /
1500                                     pose_temperature)
1501
1502         # Penalize for large actions (to promote smoother movement)
1503         action_penalty = torch.sum(torch.abs(actions), dim=1)
1504         action_temperature = 0.25 # slightly adjusted for more
1505         prominent constraint
1506         action_penalty = torch.exp(-action_penalty / action_temperature
1507                                     )
1508
1509         # Combine the rewards and penalties into a single reward
1510         total_reward = (
1511             velocity_reward +
1512             0.5 * upright_reward -
1513             0.015 * force_penalty - # slight increase for more impact
1514             0.015 * unnatural_pose_penalty - # slight increase for
1515             more impact
1516             0.015 * action_penalty # slight increase for more impact
1517         )
1518
1519         # Return the total reward and each component for analysis
1520         reward_components = {

```

```

1512
1513     "velocity_reward": velocity_reward,
1514     "upright_reward": upright_reward,
1515     "force_penalty": force_penalty,
1516     "unnatural_pose_penalty": unnatural_pose_penalty,
1517     "action_penalty": action_penalty
1518 }
1519
1520     return total_reward, reward_components
1521
1522
1523 Humanoid Task: Reward Function with best RTS (6.813) of Iteration 4(not chosen by volunteer)
1524

```

```

1525 def compute_reward(
1526     velocity: torch.Tensor,
1527     dof_pos: torch.Tensor,
1528     dof_force_tensor: torch.Tensor,
1529     actions: torch.Tensor,
1530     up_vec: torch.Tensor,
1531     heading_vec: torch.Tensor
1532 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1533     # Reward for forward velocity (run as fast as possible)
1534     velocity_reward = velocity[:, 0]
1535     velocity_temperature = 1.15
1536     velocity_reward = torch.exp(velocity_reward / velocity_temperature) - 1
1537
1538     # Encouragement for upright pose (penalize for deviation from vertical)
1539     upright_reward = up_vec[:, 2]
1540     upright_temperature = 0.55
1541     upright_reward = torch.exp(upright_reward / upright_temperature) - 1
1542
1543     # Penalize high force usage (energy efficiency)
1544     force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)
1545     force_temperature = 0.12
1546     force_penalty = torch.exp(-force_penalty / force_temperature)
1547
1548     # Penalize unnatural joint positions (for instance, avoid extreme angles)
1549     unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)
1550     pose_temperature = 0.18
1551     unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty / pose_temperature)
1552
1553     # Penalize for large actions (to promote smoother movement)
1554     action_penalty = torch.sum(torch.abs(actions), dim=1)
1555     action_temperature = 0.18
1556     action_penalty = torch.exp(-action_penalty / action_temperature)
1557
1558     # Combine the rewards and penalties into a single reward
1559     total_reward = (
1560         velocity_reward +
1561         0.5 * upright_reward -
1562         0.02 * force_penalty -
1563         0.02 * unnatural_pose_penalty -
1564         0.02 * action_penalty
1565     )

```

```
1566  
1567     # Return the total reward and each component for analysis  
1568     reward_components = {  
1569         "velocity_reward": velocity_reward,  
1570         "upright_reward": upright_reward,  
1571         "force_penalty": force_penalty,  
1572         "unnatural_pose_penalty": unnatural_pose_penalty,  
1573         "action_penalty": action_penalty  
1574     }  
1575  
1576     return total_reward, reward_components
```

Humanoid Task: Reward Function chosen by volunteer with RTS (6.983) of Iteration 5

```
1580
1581     def compute_reward(
1582         velocity: torch.Tensor,
1583         dof_pos: torch.Tensor,
1584         dof_force_tensor: torch.Tensor,
1585         actions: torch.Tensor,
1586         up_vec: torch.Tensor,
1587         heading_vec: torch.Tensor
1588     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1589         # Adjusted parameters based on analysis
1590         velocity_temperature = 1.1
1591         upright_temperature = 0.65
1592         force_temperature = 0.25
1593         pose_temperature = 0.3
1594         action_temperature = 0.3
1595
1596
1597         # Reward for forward velocity (run as fast as possible)
1598         velocity_reward = velocity[:, 0]
1599         velocity_reward = torch.exp(velocity_reward /
1600         velocity_temperature) - 1
1601
1602
1603         # Encouragement for upright pose (penalize for deviation from
1604         # vertical)
1605         upright_reward = up_vec[:, 2]
1606         upright_reward = torch.exp(upright_reward / upright_temperature
1607         ) - 1
1608
1609
1610         # Penalize high force usage (energy efficiency)
1611         force_penalty = torch.sum(torch.abs(dof_force_tensor), dim=1)
1612         force_penalty = torch.exp(-force_penalty / force_temperature)
1613
1614
1615         # Penalize unnatural joint positions (for instance, avoid
1616         # extreme angles)
1617         unnatural_pose_penalty = torch.sum(torch.abs(dof_pos), dim=1)
1618         unnatural_pose_penalty = torch.exp(-unnatural_pose_penalty /
1619         pose_temperature)
1620
1621
1622         # Penalize for large actions (to promote smoother movement)
1623         action_penalty = torch.sum(torch.abs(actions), dim=1)
1624         action_penalty = torch.exp(-action_penalty / action_temperature
1625         )
1626
1627
1628         # Combine the rewards and penalties into a single reward
1629         total_reward = (
1630             velocity_reward +
1631             0.5 * upright_reward -
1632             0.02 * force_penalty -
1633             0.02 * unnatural_pose_penalty -
1634             0.02 * action_penalty
1635         )
1636
1637
1638         return total_reward
```

```

1620
1621         0.02 * action_penalty
1622     )
1623
1624     # Return the total reward and each component for analysis
1625     reward_components = {
1626         "velocity_reward": velocity_reward,
1627         "upright_reward": upright_reward,
1628         "force_penalty": force_penalty,
1629         "unnatural_pose_penalty": unnatural_pose_penalty,
1630         "action_penalty": action_penalty
1631     }
1632
1633     return total_reward, reward_components
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673

```

### A.6.3 HUMANOIDJUMP TASK

In our study, we introduced a novel task: *HumanoidJump*, with the task description being “to make humanoid jump like a real human.” The prompt of environment context in this task is shown in Prompt 5.

#### Prompt 5: Prompts of Environment Context in *HumanoidJump* Task

```

class HumanoidJump(VecTask):
    """Rest of the environment definition omitted."""
    def compute_observations(self):
        self.gym.refresh_dof_state_tensor(self.sim)
        self.gym.refresh_actor_root_state_tensor(self.sim)
        self.gym.refresh_force_sensor_tensor(self.sim)
        self.gym.refresh_dof_force_tensor(self.sim)

        self.obs_buf[:, self.torso_position[:, :],
        self.prev_torso_position[:, :], self.velocity_world[:, :],
        self.angular_velocity_world[:, :], self.velocity_local[:, :],
        self.angular_velocity_local[:, :], self.up_vec[:, :],
        self.heading_vec[:, :], self.right_leg_contact_force[:, :],
        self.left_leg_contact_force[:, :] = \
            compute_humanoid_jump_observations(
                self.obs_buf, self.root_states[:, :],
                self.torso_position[:, :],
                self.inv_start_rot, self.dof_pos, self.dof_vel,
                self.dof_force_tensor, self.dof_limits_lower,
                self.dof_limits_upper, self.dof_vel_scale,
                self.vec_sensor_tensor, self.actions,
                self.dt, self.contact_force_scale,
                self.angular_velocity_scale,
                self.basis_vec0, self.basis_vec1)

    def compute_humanoid_jump_observations(obs_buf, root_states, torso_position, inv_start_rot,
                                           dof_pos, dof_vel, dof_force, dof_limits_lower, dof_limits_upper, dof_vel_scale,
                                           sensor_force_torques, actions, dt, contact_force_scale, angular_velocity_scale,
                                           basis_vec0, basis_vec1):
        # type: (Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, float,
        #       Tensor, Tensor, float, float, float, Tensor, Tensor) -> Tuple[Tensor, Tensor, Tensor,
        #       Tensor, Tensor, Tensor, Tensor, Tensor, Tensor]
        prev_torso_position_new = torso_position.clone()

        torso_position = root_states[:, 0:3]
        torso_rotation = root_states[:, 3:7]
        velocity_world = root_states[:, 7:10]
        angular_velocity_world = root_states[:, 10:13]

        torso_quat, up_proj, up_vec, heading_vec = compute_heading_and_up_vec(
            torso_rotation, inv_start_rot, basis_vec0, basis_vec1, 2)

        velocity_local, angular_velocity_local, roll, pitch, yaw = compute_rot_new(
            torso_quat, velocity_world, angular_velocity_world)

        roll = normalize_angle(roll).unsqueeze(-1)
        yaw = normalize_angle(yaw).unsqueeze(-1)
        dof_pos_scaled = unscale(dof_pos, dof_limits_lower, dof_limits_upper)
        scale_angular_velocity_local = angular_velocity_local * angular_velocity_scale

```

```

1674     obs = torch.cat((root_states[:, 0:3].view(-1, 3), velocity_local,
1675                        scale_angular_velocity_local,
1676                        yaw, roll, up_proj.unsqueeze(-1),
1677                        dof_pos_scaled, dof_vel * dof_vel_scale,
1678                        dof_force * contact_force_scale,
1679                        sensor_force_torques.view(-1, 12) * contact_force_scale,
1680                        actions), dim=-1)
1681
1682     right_leg_contact_force = sensor_force_torques[:, 0:3]
1683     left_leg_contact_force = sensor_force_torques[:, 6:9]
1684
1685     abdomen_y_pos = dof_pos[:, 0]
1686     abdomen_z_pos = dof_pos[:, 1]
1687     abdomen_x_pos = dof_pos[:, 2]
1688     right_hip_x_pos = dof_pos[:, 3]
1689     right_hip_z_pos = dof_pos[:, 4]
1690     right_hip_y_pos = dof_pos[:, 5]
1691     right_knee_pos = dof_pos[:, 6]
1692     right_ankle_x_pos = dof_pos[:, 7]
1693     right_ankle_y_pos = dof_pos[:, 8]
1694     left_hip_x_pos = dof_pos[:, 9]
1695     left_hip_z_pos = dof_pos[:, 10]
1696     left_hip_y_pos = dof_pos[:, 11]
1697     left_knee_pos = dof_pos[:, 12]
1698     left_ankle_x_pos = dof_pos[:, 13]
1699     left_ankle_y_pos = dof_pos[:, 14]
1700     right_shoulder1_pos = dof_pos[:, 15]
1701     right_shoulder2_pos = dof_pos[:, 16]
1702     right_elbow_pos = dof_pos[:, 17]
1703     left_shoulder1_pos = dof_pos[:, 18]
1704     left_shoulder2_pos = dof_pos[:, 19]
1705     left_elbow_pos = dof_pos[:, 20]
1706
1707     right_shoulder1_action = actions[:, 15]
1708     right_shoulder2_action = actions[:, 16]
1709     right_elbow_action = actions[:, 17]
1710     left_shoulder1_action = actions[:, 18]
1711     left_shoulder2_action = actions[:, 19]
1712     left_elbow_action = actions[:, 20]
1713
1714
1715     return obs, torso_position, prev_torso_position_new, velocity_world,
1716            angular_velocity_world, velocity_local, scale_angular_velocity_local,
1717            up_vec, heading_vec, right_leg_contact_force, left_leg_contact_force

```

**Reward functions.** We show the reward functions in a trial that successfully evolved a human-like jump: bending both legs to jump. Initially, the reward function focused on encouraging vertical movement while penalizing horizontal displacement, high contact force usage, and improper joint movements. Over time, the scaling factors for the rewards and penalties were gradually adjusted by changing the temperature parameters in the exponential scaling. These adjustments aimed to enhance the model’s sensitivity to different movement behaviors. For example, the vertical movement reward’s temperature was reduced, leading to more precise rewards for positive vertical movements. Similarly, the horizontal displacement penalty was fine-tuned by modifying its temperature across iterations, either decreasing or increasing the penalty’s impact on lateral movements. The contact force penalty evolved by decreasing its temperature to penalize excessive force usage more strongly, especially in the later iterations, making the task more sensitive to leg contact forces. Finally, the joint usage reward was refined by adjusting the temperature to either encourage or discourage certain joint behaviors, with more focus on leg extension and contraction patterns. Overall, the changes primarily revolved around adjusting the sensitivity of different components, refining the balance between rewards and penalties to better align the humanoid’s behavior with the desired jumping performance.

## HumanoidJump Task: Reward Function of Iteration 1

```
def compute_reward(torso_position: torch.Tensor,
                  prev_torso_position: torch.Tensor, velocity_world: torch.Tensor,
                  right_leg_contact_force: torch.Tensor,
                  left_leg_contact_force: torch.Tensor, dof_pos: torch.Tensor) ->
    Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Ensure all tensors are on the same device
```

```

1728     device = torso_position.device
1729
1730
1731     # Compute vertical torso movement reward
1732     vertical_movement = torso_position[:, 2] - prev_torso_position
1733     [ :, 2]
1734     vertical_movement_reward = torch.clamp(vertical_movement, min
1735     =0.0) # Reward positive vertical movement
1736     vertical_movement_reward = torch.exp(vertical_movement_reward /
1737     0.1) # Use exponential scaling with temperature
1738
1739     # Compute horizontal displacement penalty
1740     horizontal_displacement = torch.sum(torch.abs(torso_position[:, :
1741     2] - prev_torso_position[:, :2]), dim=-1)
1742     horizontal_displacement_penalty = torch.exp(-
1743     horizontal_displacement / 0.1) # Penalize large movements with
1744     temperature
1745
1746     # Compute leg forces usage reward
1747     contact_force_usage = torch.sum(torch.abs(
1748     right_leg_contact_force) + torch.abs(left_leg_contact_force),
1749     dim=-1)
1750     contact_force_usage_penalty = torch.exp(-contact_force_usage /
1751     10.0) # Penalize high contact force usage with temperature
1752
1753     # Compute joint usage reward (encourages proper leg extension
1754     and contraction)
1755     leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device
1756     =device) # Indices of leg joints
1757     leg_joint_usage = torch.mean(dof_pos[:, leg_joints_indices],
1758     dim=-1)
1759     leg_joint_usage_reward = torch.exp(-torch.abs(leg_joint_usage))
1760     / 0.1) # Encourage movements from neutral position
1761
1762     # Sum all rewards and penalties
1763     total_reward = vertical_movement_reward +
1764     horizontal_displacement_penalty + contact_force_usage_penalty +
1765     leg_joint_usage_reward
1766
1767     # Create a dictionary for individual reward components
1768     reward_components = {
1769         'vertical_movement_reward': vertical_movement_reward,
1770         'horizontal_displacement_penalty':
1771             horizontal_displacement_penalty,
1772         'contact_force_usage_penalty': contact_force_usage_penalty,
1773         'leg_joint_usage_reward': leg_joint_usage_reward
1774     }
1775
1776     return total_reward, reward_components

```

### HumanoidJump Task: Reward Function of Iteration 2

```

1774     def compute_reward(
1775         torso_position: torch.Tensor,
1776         prev_torso_position: torch.Tensor,
1777         velocity_world: torch.Tensor,
1778         right_leg_contact_force: torch.Tensor,
1779         left_leg_contact_force: torch.Tensor,
1780         dof_pos: torch.Tensor
1781     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1782         # Ensure all tensors are on the same device

```

```

1782
1783     device = torso_position.device
1784
1785     # Compute vertical torso movement reward
1786     vertical_movement = torso_position[:, 2] - prev_torso_position
1787     [ :, 2]
1788     vertical_movement_reward = torch.clamp(vertical_movement, min
1789     =0.0) # Reward positive vertical movement
1790     temperature_vertical = 0.05
1791     vertical_movement_reward = torch.exp(vertical_movement_reward /
1792     temperature_vertical) # Use exponential scaling
1793
1794     # Compute horizontal displacement penalty
1795     horizontal_displacement = torch.sum(torch.abs(torso_position[:, :
1796     :2] - prev_torso_position[:, :2]), dim=-1)
1797     temperature_horizontal = 0.1
1798     horizontal_displacement_penalty = torch.exp(-
1799     horizontal_displacement / temperature_horizontal) # Penalize
1800     large movements with temperature
1801
1802     # Compute leg forces usage reward
1803     contact_force_usage = torch.sum(torch.abs(
1804     right_leg_contact_force) + torch.abs(left_leg_contact_force),
1805     dim=-1)
1806     temperature_force = 1.0
1807     contact_force_usage_penalty = torch.exp(-contact_force_usage /
1808     temperature_force) # Penalize high contact force usage with
1809     temperature
1810
1811     # Compute joint usage reward (encourages proper leg extension
1812     and contraction)
1813     leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device
1814     =device) # Indices of leg joints
1815     leg_joint_usage = torch.mean(dof_pos[:, leg_joints_indices],
1816     dim=-1)
1817     temperature_joints = 0.05
1818     leg_joint_usage_reward = torch.exp(-torch.abs(leg_joint_usage) /
1819     temperature_joints) # Encourage movements from the neutral
1820     position
1821
1822     # Sum all rewards and penalties
1823     total_reward = vertical_movement_reward +
1824     horizontal_displacement_penalty + contact_force_usage_penalty +
1825     leg_joint_usage_reward
1826
1827     # Create a dictionary for individual reward components
1828     reward_components = {
1829         'vertical_movement_reward': vertical_movement_reward,
1830         'horizontal_displacement_penalty':
1831             horizontal_displacement_penalty,
1832         'contact_force_usage_penalty': contact_force_usage_penalty,
1833         'leg_joint_usage_reward': leg_joint_usage_reward
1834     }
1835
1836     return total_reward, reward_components

```

## HumanoidJump Task: Reward Function of Iteration 3

```

1833     def compute_reward(
1834         torso_position: torch.Tensor,
1835         prev_torso_position: torch.Tensor,

```

```

1836
1837     velocity_world: torch.Tensor,
1838     right_leg_contact_force: torch.Tensor,
1839     left_leg_contact_force: torch.Tensor,
1840     dof_pos: torch.Tensor
1841 ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1842     # Ensure all tensors are on the same device
1843     device = torso_position.device
1844
1845     # Compute vertical torso movement reward
1846     vertical_movement = torso_position[:, 2] - prev_torso_position
1847     [ :, 2]
1848     vertical_movement_reward = torch.clamp(vertical_movement, min
1849     =0.0) # Reward positive vertical movement
1850     temperature_vertical = 0.05
1851     vertical_movement_reward = torch.exp(vertical_movement_reward /
1852     temperature_vertical) # Use exponential scaling
1853
1854     # Compute horizontal displacement penalty
1855     horizontal_displacement = torch.sum(torch.abs(torso_position[:, :
1856     :2] - prev_torso_position[:, :2]), dim=-1)
1857     temperature_horizontal = 0.05 # Adjusted temperature for finer
1858     tuning
1859     horizontal_displacement_penalty = torch.exp(-
1860     horizontal_displacement / temperature_horizontal) # Penalize
1861     large movements
1862
1863     # Compute leg forces usage reward
1864     contact_force_usage = torch.sum(torch.abs(
1865     right_leg_contact_force) + torch.abs(left_leg_contact_force),
1866     dim=-1)
1867     temperature_force = 5.0 # Adjusted to make contact force usage
1868     more noticeable
1869     contact_force_usage_penalty = torch.exp(-contact_force_usage /
1870     temperature_force) # Penalize high contact force usage
1871
1872     # Compute joint usage reward (encourages proper leg extension
1873     and contraction)
1874     leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device
1875     =device) # Indices of leg joints
1876     leg_joint_usage = torch.mean(dof_pos[:, leg_joints_indices],
1877     dim=-1)
1878     temperature_joints = 0.05
1879     leg_joint_usage_reward = torch.exp(-torch.abs(leg_joint_usage)
1880     / temperature_joints) # Encourage movements from the neutral
1881     position
1882
1883     # Sum all rewards and penalties
1884     total_reward = vertical_movement_reward +
1885     horizontal_displacement_penalty + contact_force_usage_penalty +
1886     leg_joint_usage_reward
1887
1888     # Create a dictionary for individual reward components
1889     reward_components = {
1890         'vertical_movement_reward': vertical_movement_reward,
1891         'horizontal_displacement_penalty':
1892             horizontal_displacement_penalty,
1893         'contact_force_usage_penalty': contact_force_usage_penalty,
1894         'leg_joint_usage_reward': leg_joint_usage_reward
1895     }
1896
1897     return total_reward, reward_components
1898
1899

```

```

1890
1891 HumanoidJump Task: Reward Function of Iteration 4
1892
1893     def compute_reward(
1894         torso_position: torch.Tensor,
1895         prev_torso_position: torch.Tensor,
1896         velocity_world: torch.Tensor,
1897         right_leg_contact_force: torch.Tensor,
1898         left_leg_contact_force: torch.Tensor,
1899         dof_pos: torch.Tensor
1900     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1901         # Ensure all tensors are on the same device
1902         device = torso_position.device
1903
1904         # Compute vertical torso movement reward
1905         vertical_movement = torso_position[:, 2] - prev_torso_position
1906         [ :, 2]
1907         vertical_movement_reward = torch.clamp(vertical_movement, min
1908         = 0.0) # Reward positive vertical movement
1909         temperature_vertical = 0.04 # Adjusted temperature for better
1910         sensitivity
1911         vertical_movement_reward = torch.exp(vertical_movement_reward /
1912         temperature_vertical) # Use exponential scaling
1913
1914         # Compute horizontal displacement penalty
1915         horizontal_displacement = torch.sum(torch.abs(torso_position[:, :
1916         2] - prev_torso_position[:, :2]), dim=-1)
1917         temperature_horizontal = 0.1 # Increased temperature to
1918         penalize horizontal movement more
1919         horizontal_displacement_penalty = torch.exp(-
1920         horizontal_displacement / temperature_horizontal) # Penalize
1921         large movements
1922
1923         # Compute leg forces usage reward
1924         contact_force_usage = torch.sum(torch.abs(
1925             right_leg_contact_force) + torch.abs(left_leg_contact_force),
1926             dim=-1)
1927         temperature_force = 0.1 # Significantly increase sensitivity
1928         to contact forces
1929         contact_force_usage_penalty = torch.exp(-contact_force_usage /
1930         temperature_force) # Penalize high contact force usage
1931
1932         # Compute joint usage reward (encourages proper leg extension
1933         and contraction)
1934         leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device
1935         = device) # Indices of leg joints
1936         leg_joint_usage = torch.mean(dof_pos[:, leg_joints_indices],
1937             dim=-1)
1938         temperature_joints = 0.02 # Adjusted for joint usage
1939         sensitivity
1940         leg_joint_usage_reward = torch.exp(-torch.abs(leg_joint_usage) /
1941         temperature_joints) # Encourage movements from the neutral
1942         position
1943
1944         # Sum all rewards and penalties
1945         total_reward = vertical_movement_reward +
1946         horizontal_displacement_penalty + contact_force_usage_penalty +
1947         leg_joint_usage_reward
1948
1949         # Create a dictionary for individual reward components
1950         reward_components = {
1951             'vertical_movement_reward': vertical_movement_reward,
1952             'horizontal_displacement_penalty':
1953                 horizontal_displacement_penalty,
1954             'leg_joint_usage_reward': leg_joint_usage_reward
1955         }

```

```
1944
1945         'contact_force_usage_penalty': contact_force_usage_penalty,
1946         'leg_joint_usage_reward': leg_joint_usage_reward
1947     }
1948
1949     return total_reward, reward_components
1950
```

## Humanoid Task: Reward Function of Iteration 5

```
1954
1955     def compute_reward(
1956         torso_position: torch.Tensor,
1957         prev_torso_position: torch.Tensor,
1958         velocity_world: torch.Tensor,
1959         right_leg_contact_force: torch.Tensor,
1960         left_leg_contact_force: torch.Tensor,
1961         dof_pos: torch.Tensor
1962     ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
1963         # Ensure all tensors are on the same device
1964         device = torso_position.device
1965
1966         # Compute vertical torso movement reward
1967         vertical_movement = torso_position[:, 2] - prev_torso_position[:, 2]
1968         vertical_movement_reward = torch.clamp(vertical_movement, min=0.0) # Reward positive vertical movement
1969         temperature_vertical = 0.04 # Adjusted temperature for better sensitivity
1970         vertical_movement_reward = torch.exp(vertical_movement_reward / temperature_vertical) # Use exponential scaling
1971
1972         # Compute horizontal displacement penalty
1973         horizontal_displacement = torch.sum(torch.abs(torso_position[:, :2] - prev_torso_position[:, :2]), dim=-1)
1974         temperature_horizontal = 0.05 # Decreased temperature for more sensitivity
1975         horizontal_displacement_penalty = torch.exp(-horizontal_displacement / temperature_horizontal) # Penalize large movements
1976
1977         # Compute leg forces usage penalty (Rewritten to reduce contact force)
1978         contact_force_usage = torch.sum(torch.abs(right_leg_contact_force) + torch.abs(left_leg_contact_force), dim=-1)
1979         temperature_force = 0.5 # Adjusted to penalize contact force usage
1980         contact_force_usage_penalty = torch.exp(-contact_force_usage / temperature_force) # Penalize high contact force usage
1981
1982         # Compute joint usage reward (encourages proper leg extension and contraction)
1983         leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device=device) # Indices of leg joints
1984         leg_joint_usage = torch.mean(torch.abs(dof_pos[:, leg_joints_indices]), dim=-1)
1985         temperature_joints = 0.02 # Adjusted for joint usage sensitivity
1986         leg_joint_usage_reward = torch.exp(-leg_joint_usage / temperature_joints) # Encourage movements from the neutral position
```

```

1998
1999     # Sum all rewards and penalties
2000     total_reward = vertical_movement_reward +
2001     horizontal_displacement_penalty + contact_force_usage_penalty +
2002     leg_joint_usage_reward
2003
2004     # Create a dictionary for individual reward components
2005     reward_components = {
2006         'vertical_movement_reward': vertical_movement_reward,
2007         'horizontal_displacement_penalty':
2008             horizontal_displacement_penalty,
2009         'contact_force_usage_penalty': contact_force_usage_penalty,
2010         'leg_joint_usage_reward': leg_joint_usage_reward
2011     }
2012
2013
2014     return total_reward, reward_components
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051

```

## Humanoid Task: Reward Function of Iteration 6

```

def compute_reward(
    torso_position: torch.Tensor,
    prev_torso_position: torch.Tensor,
    velocity_world: torch.Tensor,
    right_leg_contact_force: torch.Tensor,
    left_leg_contact_force: torch.Tensor,
    dof_pos: torch.Tensor
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Ensure all tensors are on the same device
    device = torso_position.device

    # Compute vertical torso movement reward
    vertical_movement = torso_position[:, 2] - prev_torso_position[:, 2]
    vertical_movement_reward = torch.clamp(vertical_movement, min=0.0) # Reward positive vertical movement
    temperature_vertical = 0.03 # Fine-tuned temperature for better sensitivity
    vertical_movement_reward = torch.exp(vertical_movement_reward / temperature_vertical) # Use exponential scaling

    # Compute horizontal displacement penalty
    horizontal_displacement = torch.sum(torch.abs(torso_position[:, :2] - prev_torso_position[:, :2]), dim=-1)
    temperature_horizontal = 0.04 # Decreased temperature for more sensitivity
    horizontal_displacement_penalty = torch.exp(-horizontal_displacement / temperature_horizontal) # Penalize large movements

    # Compute leg forces usage penalty (encourage minimal contact force)
    contact_force_usage = torch.sum(torch.abs(right_leg_contact_force) + torch.abs(left_leg_contact_force), dim=-1)
    temperature_force = 0.5 # Adjusted to penalize contact force usage
    contact_force_usage_penalty = torch.exp(-contact_force_usage / temperature_force) # Penalize high contact force usage

    # Compute joint usage reward (encourages proper leg extension and contraction)

```

```

2052
2053     leg_joints_indices = torch.tensor([6, 7, 8, 12, 13, 14], device
2054         =device) # Indices of leg joints
2055     leg_joint_usage = torch.mean(torch.abs(dof_pos[:, leg_joints_indices]), dim=-1)
2056     temperature_joints = 0.02 # Fine-tuned for joint usage
2057     sensitivity
2058     leg_joint_usage_reward = torch.exp(-torch.abs(leg_joint_usage)
2059         / temperature_joints) # Encourage movements from the neutral
2060         position
2061
2062     # Sum all rewards and penalties
2063     total_reward = vertical_movement_reward +
2064         horizontal_displacement_penalty + contact_force_usage_penalty +
2065         leg_joint_usage_reward
2066
2067     # Create a dictionary for individual reward components
2068     reward_components = {
2069         'vertical_movement_reward': vertical_movement_reward,
2070         'horizontal_displacement_penalty':
2071             horizontal_displacement_penalty,
2072         'contact_force_usage_penalty': contact_force_usage_penalty,
2073         'leg_joint_usage_reward': leg_joint_usage_reward
2074     }
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105

```